A Virtual Distributed Computing System

by

Mohammed Ghouseuddin

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

March, 1998

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

A VIRTUAL DISTRIBUTED COMPUTING SYSTEM

BY

MOHAMMED GHOUSEUDDIN

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In
COMPUTER SCIENCE

MARCH 1998

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DHAHRAN 31261, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by **MOHAMMED GHOUSEUDDIN** under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE.**

THESIS COMMITTEE

Dr. Muslim Bozyigit (Thesis Advisor)

Dr. Jarallah S – AlGhamdi (Member)

Dr. Hassan Barada (Member)

Department Chairman

Dean, College of Graduate Studies

11/5/98
Date

Dedicated to

Ammi, Abba

and

Ahmedi, Ghayas, Saleha, Azgary

ACKNOWLEDGEMENTS

In the name of Allah, Most Gracious, Most Merciful.

All Praise is due to ALLAH to whom belongs the dominion of the Heavens and the Earth. Peace and mercy be upon His Prophet. I thank Him for giving me the knowledge and patience to carry out this work.

First and foremost, I would like to express my humble gratitudes to my Ammi & Abba, without whose blessings, prayers and motivation, I wouldn't have been what I am. I would like to take this oppurtunity to acknowledge my sisters and brothers whose love and encouragement allowed me in pursuing my studies here at KFUPM, far away from them.

I would like to offer my indebtness and sincere appreciation to my thesis advisor Dr. Muslim Bozyigit and no word of thanks would be sufficient for his guidance. He was always available and ready to help me. Special thanks are due to my thesis committee members Dr. Jaralla Al-Ghamdi and Dr. Hassan Barada for all their cooperation and advice.

Thanks are due to my friends Aiyaz, Irfan, Jameel, shahid and all others with whom I have spent some memorable moments.

Finally, acknowledgement is due to King Fahd University of Petroleum and Minerals for the support and resources provided in this research.

Mohammed Ghouseuddin.

Contents

List of Tables

List of Figures

List of Algorithms

THESIS ABSTRACT

Name: MOHAMMED GHOUSEUDDIN

Title: A VIRTUAL DISTRIBUTED COMPUTING SYSTEM

Degree: MASTER OF SCIENCE

Major: COMPUTER SCIENCE

Date of Degree: MARCH 1998

Cost of parallel and closely coupled architectures and the complex issues involved in parallel computing have led to the alteration in the state of computing. With the advent of high performance workstations and high speed networking, the computational power of workstations have become attractive and indispensable for parallel computing. Utilizing the idle CPU cycles of a remote processor, if the jobs cannot be processed locally, is the current trend in research.

The main objective of this study is to transform a network of workstations into a Virtual Distributed Computing System(VDCS). Such a system has two components: load balancing and fault tolerance. Load balancing in a distributed computing system improves the performance of the system to a substantial amount while improving the job response times. Defining the load metrics, determining the load on individual hosts, detecting the load imbalance and load balancing the system are the issues to be dealt in load balancing. Providing backup for each application, detecting host failures, restoring the failed applications are the issues involved in fault tolerance.

Performance analysis of the system has been done using a number of hypothetical applications and one real application(in this case matrix multiplication). Hypothetical applications provide a flexibility for tuning the environment to test VDCS. VDCS has responded positively to parallel matrix multiplication. Using load balancing the average speedup achieved for different applications was 66% of its theoritical level, on a small network of workstations, while fault tolerance has proved to provide a reliable environment.

Keywords: Network of workstations,dynamic load balancing, parallel processing, fault tolerance, task migration.

King Fahd University of Petroleum and Minerals, Dhahran.
March 1998

خلاصة الرسالة

اسم الطالب : محمد غوث الدين

عنوان الرسالة : نظام افتراضي للعمليات الحسابية الموزعة

التخصص : الحاسب الآلي

تاريخ الشهادة : مارس ١٩٩٨م

إن التكاليف الباهضة لإجراء العمليات بواسطة الأنظمة المتوازية والمزدوجة ، إضافة إلى المواضيع المعقدة من نظام الحسابات المتوازية ، فرضت وجود نظام بديل لإجراء الحسابات . ومع تطوير محطات عمل ذات فعالية عالية ووجود شبكات حاسبات سريعة ، فإن القدرة الحسابية العالية لمحطات العمل أصبحت أمراً مرغوباً وضرورياً للأنظمة الحسابية المتوازية . وتتجه البحوث العلمية في الوقت الحاضر نحو استغلال بدورات وحدة الحسابات المركزية المهددة في معالج آخر بعيد – اذا كان من المتعسر إجراء العملية من المعالج المحلي .

إن الهدف الرئيسي من هذه الرسالة هو تحويل شبكة محطات العمل إلى نظام إفتراضي للعمليات الحسابية الموزعة . إن هذا النظام يحتوي على عنصرين : موازنة الأحمال . والصمود أمام الأعطال ، وعند تطبيق موازنة الأحمال في أي نظام توزيعي فإنها ترفع من فعالية النظام لدرجة ملحوظة كما أنها تحسن زمن استجابة العمل للنظام . ومن ضمن المواضيع المتعلقة بموازنة الأحمال : تعريف مصفوفة الأحمال ، تحديد الحمل في كل جهاز مضيف ، كشف الاختلال في الأحمال وموازنتها . كما أن من المواضيع المتعلقه في مجال صمود النظام أمام الأعطال ما يلي : عمل نسخة احتياطية لكل عمل ، والكشف عن الجهاز الذي أصابه خلل ، وإعادة تشغيل التطبيقات المتوقعة نتيجة العطل .

لقد تم إجراء اختبار فعالية النظام بإستخدام عدد من التطبيقات الإفتراضية وكذلك استخدم تطبيق حقيقي (وهو ضرب المصفوفات) ، عند استخدام التطبيقات الافتراضية فإنه تساعد في إجراء تغييرات حسب الحاجة لقياس فعالية النظام وكانت نتجية الإختبارات ايجابية في حالة تطبيقات ضرب المصفوفات . ونتجية وجود موازنة الأحمال فإنه كان بالأمكان الحصول على تسارع بمعدل ٢ للتطبيقات المختلفة في حال استعمال ٣ محطات عمل في النظام ، وكذلك فإن النظام أثبت جدارته في توفير بيئة صامدة أمام الأعطال .

المصطلحات : شبكة محطات العمل ، موازنة الأحمال ، تطبيقات متوازية ، الصمود أمام الأعطال ، ارتجال التطبيقات .

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

الظهران – المملكة العربية السعودية

مارس ١٩٩٨م

Chapter 1

Introduction

Tremendous increase in computing and communication speeds have been achieved in the past four decades due to the advances in hardware and software technology. The achievements have led to the development of parallel and distributed architectures. Personal computers and workstations have become a new trend, which coupled with advances in communication technology have altered the state of computing.

A Distributed Computing System(DCS) is a concept of the 90's which refers to a collection of more than one loosely coupled computers connected by a data communication network. The scope of a DCS covers clusters of local area networks. The workstations may be homogeneous or heterogeneous. The largest example of such a DCS is an internet, comprising hundreds of thousands of computers ranging from PC's to Mainframes to Super computers connected by a global network.

A DCS is largely suitable for client-server type applications like network file

systems and distributed database systems. Numerical oriented applications like weather forecasting, guessing game problems, image processing, and iterative algorithms are few of the areas which will have a improved performance if run on DCS.

Workstation based DCS have become popular in both academic and commercial communities due to the continuing trend of decreasing cost/performance ratio. The processors communicate by message passing. Loosely coupled parallel computing systems have become famous due to the fact that the tightly coupled parallel systems are costly and they are suited to a limited set of applications.

1.1 Resource utilization and reliability in DCS

In a typical DCS, the probability that some nodes are idle while some other nodes are heavily loaded is high, and such situations can be remedied by balancing the loads on the nodes in the system. Though the demand for extra computing power is never ending, users would like to utilize the cpu cycles of an idling or under utilized machine if their jobs cannot be processed efficiently on their local workstations.

The issue of how to effectively utilize the computing power in workstation based distributed systems has sparkled many research ideas and experimental systems. Most of the work concentrates on topics of analyzing workstation usage patterns, designing algorithms for remote capacity utilization, and developing facilities for

remote execution.

Another important issue is the reliability i.e., fault tolerant aspect of the applications and systems. The reliability issue comes into focus if we consider the machine failures and also the ownership of the personal workstations. A personal workstation autonomy is kept within the hands of their owners. A load balanced job should be preempted from the workstation whenever the owner needs to use the machine.

1.2 Motivation and Objectives

1.2.1 Motivation

Parallel processing has been a major trend of the 90's. The inherent complexity involved with the hardware and the software of parallel systems makes them very costly.

A DCS will be in a state of imbalance if some workstations are heavily loaded and some are lightly loaded, which may degrade the performance of the system. Improving the job response time and utilizing the system resources efficiently and providing a reliable environment for the jobs is a potential research direction.

The issue "Effective utilization of system resources" has led to the concept of Load Balancing Subsystem. The issue of providing application reliability has led the Fault Tolerance Subsystem. To provide load balancing we have to consider the

following issues.

- Finding whether the system is suffering from a load imbalance.
- Determining the workstation load indices used to measure the load on a workstation.
- Finding the workstations which are heavily loaded and the ones that are idle.
- Determining the jobs that are suitable for load balancing.
- Identifying the suitable processors to execute the local jobs remotely.
- Choosing the right load balancing policy (Centralized vs Distributed).
- The above systems have to be integrated into LBFTS(Load Balancing and Fault Tolerance Subsystem).

To visualize the importance of a providing a fail safe environment a narrow scope study was conducted on our departmental DCS [1]. Of the 200 workstations comprising the DCS a set of about 35 machines were selected at random and the number of failures of the machines over a period of 30 days were recorded. The statistics obtained are shown as a graph in Figure 1.1.

A failure rate of an average of 4 machines/day was observed, and as the number of machines is increased, the rate is expected to grow linearly. A fault tolerance system can provide a fail safe environment.

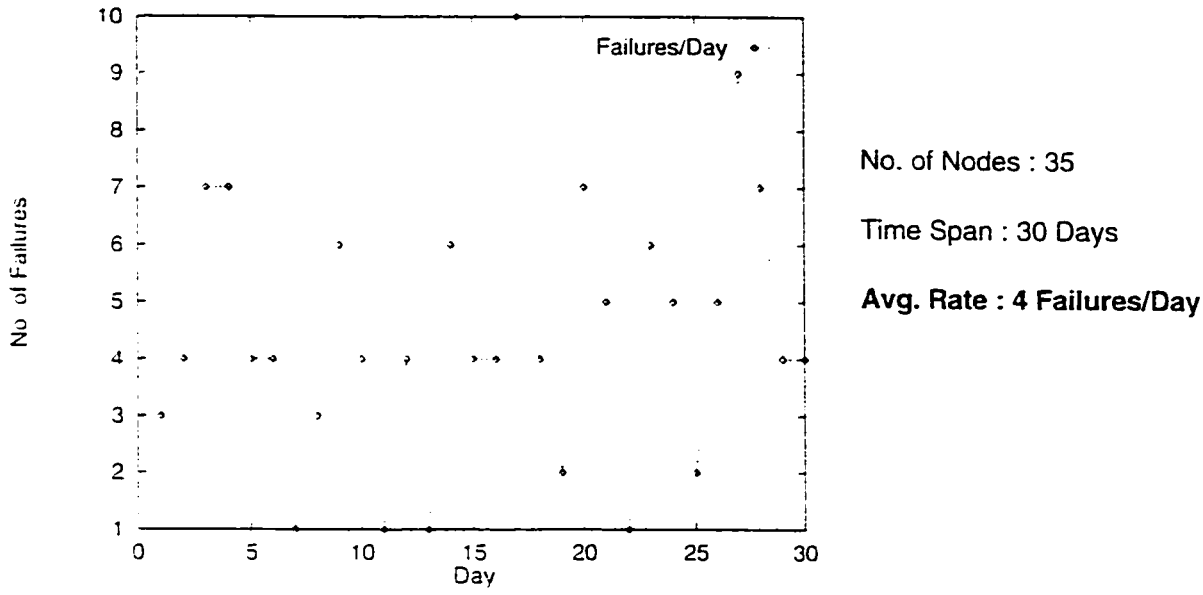


Figure 1.1: Rate of Failures

1.2.2 Problem statement

A DCS without a Load Balancing facility leads to inefficient usage of system resources. A system crash requires atleast the restart of the jobs running on the machine from the scratch if a fault tolerant system is not provided. Every host on a system should have load balancing facility and then the capability of detecting failures on other machines. The main mission of this study is to integrate a load balancing, fault tolerance and task migration subsystems into a coherent DCS.

The main objectives of this research are stated as:

1. Designing and Implementing a dynamic load balancing subsystem. The framework is to be developed using effective load indices and algorithms.

2. Designing and implementing a fault tolerance subsystem. It should have the following capabilities.

   - Detect and report machine failures.
   - Backup each machine on the DCS.
   - Restore/Restart the jobs of failed machines.

3. Incorporate previous two subsystems into Process Migration Subsystem(PMS) [2].

4. Development of an application interface for the end system.

5. Study the performance characteristics of the system as function of various parameters, using hypothetical as well as real applications.

1.2.3 Role of the system in a DCS

The position of a load balancing and fault tolerance system(LBFTS) in a DCS is depicted in Figure 1.2.

The fault tolerance subsystem keeps track of the machine failures. The applications are submitted to the API which interacts with the load balancing subsystem. The load balancing system in turn uses the services of the PMS.

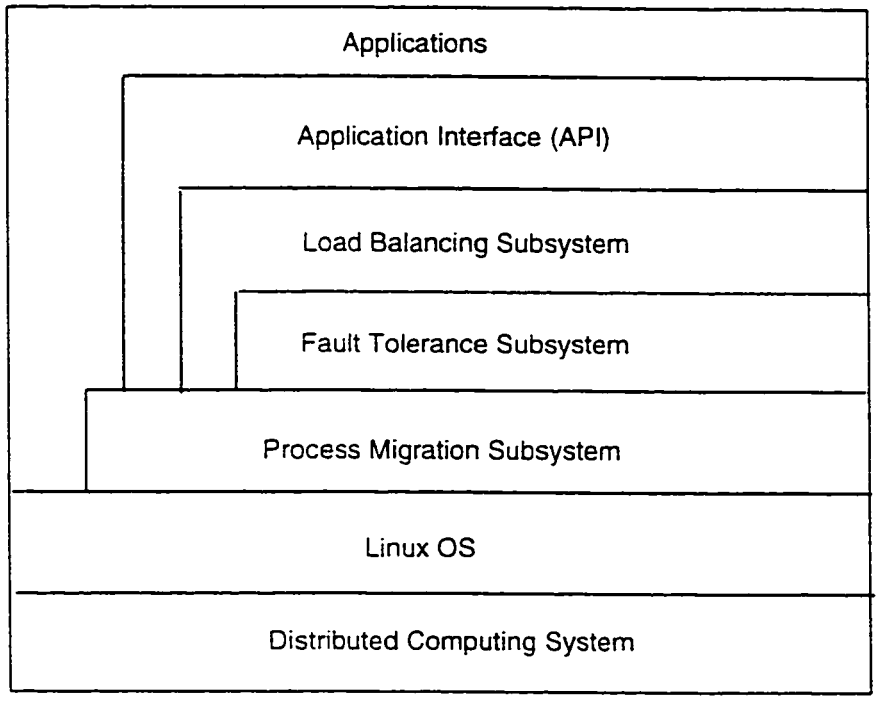


Figure 1.2: Overall System View

Chapter 2

Related Work

A DCS is a pool of resources like cpu, i/o, communication bandwidth and memory. The operating system without any external support cannot utilize the resources efficiently. Though time sharing of the cpu is a method used to provide equal share of the cpu to all processes running on the system, a particular system may not be having any processes to be serviced at some point of time; and at the same time some other machine over the network may be busy servicing its applications and hence over utilizing the cpu.

If all the resources on the network can be considered as a single pool and think of timesharing all the jobs on all the machines to this single pool then there is a fat chance of efficient utilization of the network resources. Such an approach taken in the literature is the Load Balancing. In addition to utilizing the resources efficiently, if the jobs can be provided a reliable environment so that they run to completion,

then we can say that the system will provide a fail safe environment.

2.1 Literature Survey

In this section the literature is reviewed and compared with the research work. In the first subsection the literature review is done for load balancing and in the second section the literature is reviewed for fault tolerance.

2.1.1 Load Balancing

There have been many approaches towards Load Balancing. All approaches fall into two major categories : Static and Dynamic. Static load balancing has been a state of art of the late 80's and it is not discussed here. Dynamic load balancing is the current trend for DCS. Characteristic features of Dynamic Load balancing are as stated below:

- Cost of computing a task and cost of communication between jobs is not known prior to job execution.
- Work load on a machine is the work done/to be done in a particular period of time. The numeric quantity used to measure the load on a workstation(ws) is known as a Load Index. System workload varies dynamically.
- Load balancing is done while the applications are running on the system.

According to Zhu et.al [3] load balancing can be done at task initiation or task execution. In the first policy the load balancer selects processors for newly arrived tasks. The load balancer has load information of all the processors and chooses the suitable processor to execute a newly arrived task. The policy has the disadvantage that the load balancing cannot be done efficiently. If a host detects that it is experiencing a load imbalance it has to wait till a new job is submitted to the host. In the second policy a migration mechanism is used to transfer executing tasks to other machines if a load imbalance is found in the system. The tasks chosen for transfer can be new arrivals also.

Arredondo et.al. [4] present a load distribution and balancing system for work-station based DCS. The system is a minimal part of the current research. Load balancing is done at task initiation. The load index used in this paper is a multiple resource metric(mrm) where

$$mrm =< CAPP, CMC, DT, NT > \tag{2.1}$$

where

CAPP, the current available processing power in the node,

CMC, the current memory capacity,

DT, the amount of data transferred on each disk,

NT, the number of I/O packets on each network interface.

Michael.L et.al [5] give a brief outline of the load balancing method in a public/private owned workstation environment. A long running job is assigned to an idle workstation and the job is evicted whenever the owner of the idle workstation resumes work. The authors give a general description, but do not provide the definitions for load. A centralized host is responsible for assigning the computing capacity to all workstations and each host does its own scheduling. This method is more flexible in the sense that a centralized host does not become the bottleneck.

Phillip K. and R. Chawla [6] provide a design for a distributed scheduler which uses a local scheduling mechanism known as priority resource allocation. Through priority resource allocation it insulates a workstation owner from the effects of foreign processes, while allowing the foreign processes to continue using whatever resources that are left over. A global scheduling mechanism is used to assign foreign processes to a machine. The paper does not specify the technique used to assign the foreign processes.

According to Zhu et.al [3],Zhou [7],[8] a load balancing algorithm consists of four components.

1. Information policy module does the job of keeping upto date load information of each machine.

2. Transfer policy module determines the eligibility of tasks for Load Balancing based on the task size and the load on the hosts.

3. Placement policy decides for eligible jobs for transfer and the possible destination hosts for the eligible jobs based on the load on the hosts.

4. Checking policy used to check the reachability of processors to provide fault tolerance.

The above authors provide a general outline for load balancing and fault tolerance. Design and Implementation involves many issues which are not discussed in these papers.

Willebeek and Reeves [9] support the above characteristics of a load balancing policy. In addition they add one more parameter known as load balancing profitability determination which is used to estimate the potential speedup obtainable through load balancing. The algorithms proposed in this paper are very much suitable for the hypercube networks. There has been much research on the load indices used to collect load information. The load of a system is the indicator of work being done on the system. Nisar [10],[11] uses a history based heuristic to know the CPU, I/O and memory requirements of a job.

According to Philip et.al [13],[9] the current ready queue length i.e. the number of processes in the system is used as a load index. They show that current queue length does not give a correct estimate of the load on the system as it is not suitable as a well tuned parameter for load balancing.

Current load at a node k in Erciyes and Yilmaz [14] is given by :

$$Load_k = A.n_f + B.n_p + C.n_t \tag{2.2}$$

where

n_f is the number of processes waiting at the FCFS queue,

n_p is the number of processes waiting at the priority based preemptive queues and ,

n_t is the number of processes at the round robin queue.

A, B and C are the system dependant constants which can be tuned as required. This paper considers real time jobs and assigns them a different queue for scheduling. This takes care of response critical processes. The technique used is different in the sense that it is suitable for a hard real time environment where a population of processes has a strict deadline to be met.

In [15] Hou and Shin use **state** as a Load index which is a combination of three quantities.

- AvgLoad taken as queue length over a period of time.
- KeyboardIdle time (an average of 15 minutes).
- State of the workstation which can be NoTask, TaskRunning , Suspended, Vacating or Killed.

Keyboard idle time is not suitable for machines which have background tasks. State is a volatile value which cannot be relied upon to estimate the load.

Zhu and Socko [3] discuss a number of methods to collect load information. A load balancing mechanism with task initiation only uses the formula given below to calculate the load value.

$$value(h.pd) = CPU\_speed_h / runnable_h + memory(h.pd). \tag{2.3}$$

where

h stands for the processor.

pd stands for the process.

CPU_speed_h stands for the nominal speed of the processor.

$runnable_h$ stands for the number of processes in the processor queue.

memory($h.pd$) is a parameter used to determine if processor has enough memory to execute process pd.

If load balancing with task initiation and migration is considered then load indices used are

- Queue length.
- Average CPU utilization.

Knowing the memory required by process pd is an utmost requirement to estimate whether the processor has enough memory to execute the process.

M.V.Deverakonda et.al [16].[17].[18] propose a statistical approach for predicting the CPU time , the file I/O and the memory requirements of a program at the beginning of life of the process. The steps involved in a prediction based algorithm are.

- Compute the CPU load for all the nodes.
- Feed the scheduled process to the predictor.
- Send the process to the node with the smallest CPU load.
- Update the load information.

The authors state that the proposed system predicts the exact resource requirements of a process.

According to Monteil and Garcia[19], the task allocation is done based on the predicted process execution. The process execution time is predicted using markov chain queuing model. Average system load value calculated on the previous week is used as a history which combined with the process arrival rate gives the current load on the system. According to the authors this method provides a way to predict the task allocation efficiently. The method is more theoretical and the proposed load index is not well tuned.

Another issue that has been explored in the literature is load balancing policy location and algorithms used for load balancing. Different load balancing methods have been studied and simulated and the results have been compared by Zhou [7],Erciyes et.al[14]. These methods are based on the fact that they are either centralized or distributed. The centralized algorithms have a single load controller and balancer for the complete system whereas in the case of distributed system each host has its own load collection and the load balancing mechanism. The following algorithms were discussed.

- Global: One host is designated as a Load Information Center(LIC) which receives load updates from all the other hosts and assembles them as a single vector and then broadcasts it to every host on the DCS. So each host checks the vector with the lowest load and transfers its own load if there is such a need.

- Disted: Instead of reporting the local load to a centralized LIC as in Global, each host broadcasts its load periodically for the other hosts to update their locally maintained load vector.

- Central: In this algorithm the LIC is also responsible for load balancing in addition to the load collection. If a host has to be balanced it sends a request to the LIC and gets balanced.

- Random: This algorithm uses only local load information when a job is found to be eligible for load balancing . it is sent to a randomly selected host.

- Thrshld: S number of randomly selected hosts are polled when an eligible job arrives and the job is transferred to the first host whose load is below a load threshold T. If no such host is found the job is processed locally.

- Lowest: Similar to Thrshld except that instead of using a threshold for a placement. a fixed number of hosts(L) are polled and the most lightly loaded host is selected. The probing is stopped if an idle host is found.

Disted has a drawback that it requires many messages for load updates. Random.Thrshld and Lowest do not consider the global state of the load information. These algorithms are useful when there is a need to reduce the overload due to load balancing. Central is considered as the most effective algorithm but the LIC may become a processing bottleneck. Global is the method adopted in our research. The authors simulate the algorithms. They are not implemented and tested in a real environment.

Willebeek and Reeves [9] and Tandiary et.al [20] discussed five different load balancing strategies.

- Gradient model: Every processor interacts with its nearest neighbor. Lightly loaded processors inform their neighbors their load status and the heavily loaded processors respond by sending some load to them. Here we have a

LOW water mark and a High water mark. Low water mark denotes light load and High water mark denotes a heavy load. Depending on the nearness(hop count) of the processors the load balancing is done.

- Sender initiated diffusion: A sender is an overloaded processor and the receiver is a lightly loaded processor. The sender initiates the load balancing. The strategy used by the sender involves either selecting a processor randomly without any load information exchange or selecting processors randomly and comparing their loads and transferring the load to the processor with the minimum load.

- Receiver initiated diffusion: Is the converse of the sender initiated diffusion. Here the receiver initiates the load balancing and the sender responds.

- Hierarchical Load Balancing(HBM): The multicomputer system is organized into a hierarchy of balancing domains , thereby decentralizing the control of the load balancing operations. Specific processors are designated to control the balancing operations at different levels of the hierarchy. Usually the processors are logically organized as a binary tree.

- Dimension Exchange method(DEM): This is similar to the HBM but it is a synchronous approach where different domains are balanced in a synchronized manner.

These methods are suitable for a hypercube of workstations. But the Sender initiated diffusion and the receiver initiated diffusion can be used in any type of network. These diffusions may lead to transfer of load from many heavily loaded hosts to a single lightly loaded host thus loading that host heavily.

M.Nuttal and M.Sloman [21] investigate the workload characteristics required for load balancing and process migration. It proposes a preemptive load balancing, whereby executing tasks may be migrated from one host to another. Load is calculated using the formula

$$L_i = (1, T_{cpu}/A_{cpu}, T_{lio}/A_{lio}, T_{rio}/A_{rio}). \tag{2.4}$$

where

T_{cpu} is cpu time used by process i.

T_{lio} is local I/O time used by process i.

T_{rio} is remote I/O time used by process i.

A_{cpu} is mean cpu time per process calculated on a history basis.

A_{lio} is mean local I/O time per process and

A_{rio} is mean remote I/O time per process.

The authors do not provide the details of the above load indices.This paper states that cpu bound jobs are suitable for preemption. Short lived jobs are filtered

out and only long lived jobs are considered suitable candidates. The migration is based on the ratio of the local I/O and the remote I/O done by each process.

Esquivel et.al [22] propose a sender initiated Load Balancing strategy where a sender on job initiation polls some hosts to relocate a job and on getting a reply sends the jobs to the suitable host.The load index used is the CPU queue length which is not considered to be an accurate load index. Cpu queue length shows the number of processes currently using the processor, but does not give the estimate of how much cpu is used by each process.

Zaki et.al [23] propose a dynamic load balancing algorithm for parallel applications. The balancing is done by initial placement of work to each workstation and synchronizing the workload distribution each time a particular workstation finishes. The synchronization is done by a centralized host. The system considers different parameters like processor speed, network topology, program parameters like number of iterations, work per iteration, time per iteration.

2.1.2 Fault Tolerance

Fault tolerance and reliability has been considered in the literature but mostly suitable for a particular architecture.

Anita borg et. al [24] provide a design for a distributed fault tolerance version of UNIX. In this design fault tolerance is achieved by providing a backup process on another machine for each primary process running on the system. A backup process

consists of sufficient information to begin execution in case of a machine failure. The backed up process is consistent in terms of input and output messages and the computed state and can eventually catchup and takeover as the original process. To maintain IPC between the processes the primary and the backup processes receive the same messages so that on recovery there will not be any loss of information. This system is very complex but maintains a consistent state in terms of the processing and message overhead.

In literature checkpointing has been considered a step towards fault tolerance. Checkpointing coupled with a migration facility has been the only effective method used to provide real fault tolerance. Process migration with checkpointing has been dealt in detail by Yeshayahu and Raphael [25]. Douglis and Ousterhout [26]. The authors deal with the checkpointing and the migration of checkpointed processes. But the fault tolerance issue is not dealt in specific. Detecting the faults and managing the backups and the recovery mechanism is not dealt with.

Naseer [2],[1] has implemented a Process Migration Subsystem(PMS) which provides fault tolerance by providing an environment where each process registers itself with the PMS, specifying the granularity of the reliability needed by it in the form of an interval between two successive checkpoints. The PMS takes care of the checkpointing and migration mechanisms. This system provides an infrastructure to be used by higher level systems like Load Balancer and Fault Tolerance.

Richmond and Mithchens [27] propose a new process migration algorithm which

has a improved performance over the other methods used in the literature. They compare four types of migration algorithms: post copy,precopy, oncopy and outcopy. The paper claims that process migration improves system reliability.

Plank and Li [28] claim to provide a consistent and optimized checkpointing methods to provide fault tolerance. The optimization methods used in checkpointing are main memory checkpointing and compression. In main memory checkpointing the checkpointer uses main memory to store the checkpoint and delay writing the checkpoint to the disk. It is written to the disk when the system is idle. In compression the checkpoint is compressed reducing the number of bytes written to the disk, thus reducing the I/O overhead.

Prakash and Singhal [29] propose a consistent snapshot algorithm for checkpointing. It is a nonintrusive and efficient algorithm. A consistent global snapshot is taken which consists of local states of all the nodes and the messages in transit along all the communication channels. Using this global snapshot a global recovery can be done as a roll back can be done from the previous checkpoint. Faults are detected through neighbor communication and if there is a crash the messages are discarded and a roll back message is issued. A roll back on a single machine may lead to a roll back on other machines if the jobs on the machines are related through communication. This paper focuses mainly on the interprocess communication.

Jeremy et.al [30] propose a transparent checkpointing and process migration for load balancing. This paper provides an environment for providing fault tolerance

by using checkpointing. The checkpointed files are accessed by all the workstations through a global file system. If a global file system is unavailable then the checkpointed files are stored on a centralized server.

D.Kebbal et.al [30] present a new approach for checkpointing parallel applications. The algorithm optimizes the checkpointing process and hence reduces the time required to checkpoint each and every process on the network.

The policy and algorithms used for Load Balancing and Fault tolerance will be discussed in chapter 3.

Chapter 3

Design of LBFTS

The system designed in this chapter provides a load balanced and reliable operating environment for independent and communicating applications. Equitable distribution of system resources among different user applications is the issue to be handled by the Load Balancing Subsystem. Provision for migration of jobs of failed machines is handled by the Fault Tolerance Subsystem. Finally integration of the two systems should be done in a user transparent manner.

This chapter outlines the issues involved and the difficulties encountered in developing such a system.

3.1 Requirements Specification

The LBFTS constitutes of two main subsystems as listed below.

1. Load Balancing Subsystem(LBS).

2. Fault Tolerance Subsystem(FTS).

3.1.1 Load Balancing Subsystem

The basic functionality of LBS is to distribute the load equally among all the different workstations. Load balancing of applications can be done in two ways as listed below:

- Job Entry Load Balancing(Jlb).
- Periodic Load Balancing(Plb).

Jlb is to be done when an application enters the system. It is initial placement of tasks i.e., executing them only after they are load balanced. In Periodic Load Balancing, after every fixed duration the system decides on which application to be evicted from a particular workstation(ws) and relocated to another. The above situation will arise only if the ws is heavily loaded and some other ws is lightly loaded. If the system decides to evict a particular application then it should be migrated onto the chosen machine and restored/restarted based on the availability or non-availability of the checkpointed file respectively.

The functionality of LBS is performed by different components as listed below:

1. Load Monitor(LM).

2. Application Agent(AA).

3. Application Identifier(AI).

4. History Management system(HM).

5. Load Balancer(LB).

The functionality of a LM is to collect local ws load information and inform it to the LM on a responsible ws which forms a global database of the load on all workstations. This global information is sent back to all LM's running on the workstations.

The basic functionality of an AA is monitoring individual applications and informing the LBFTS about the application entry and completion. The AA collects the resource usages of each application which is kept as resource usage history.

The responsibility of an AI is to uniquely identify applications that are executed on the system. The history provided by HM is used to predict the resource requirements of a particular application. The HM manages the history database. Application history recording and database maintenance and performing the different operations on the database are the responsibilities of a HM.

The LB uses the information provided by the above components to load balance the applications.

3.1.2 Fault Tolerance Subsystem

The basic functionality of FTS is detection and tolerance of machine failures. A ws is said to be alive if its working and accessible to other workstations over the

network, otherwise it is considered dead. Only live workstations are considered a member of the local group. The system is flexible enough to add a new ws to the group. New workstations are those that have been dead for some time or those that are added newly to the network.

The components identified in the FTS are

1. Fault Detection Mechanism(FDM).

2. Fault Recovery Mechanism(FRM).

The responsibility of Fault Detection Mechanism is to identify the faulty machines and inform the same to other live workstations. The responsibility of the Fault Recovery Mechanism is to recover the failed applications.

The services provided by FTS are to be utilized by LBS. A workstation that fails and had applications running, should be detected and the LBS should be informed. The LBS takes care of the applications by re-allocating them in a load balanced manner.

Process Migration Subsystem provides a roll back mechanism using checkpointing method. Each application running on the system is checkpointed regularly and these checkpoints are used to restore/restart applications in case of machine failures.

3.2 Design of LBS

The design and analysis of different components of LBS . the interface with the other system components and the algorithms used in the design are discussed in this section.

3.2.1 Load Monitor

The issues that arise in the design of a Load Monitor are:

1. Determination of load metrics on individual workstations.

2. Load monitoring policy.

Determination of Load Metrics

The load index used is a prediction based dynamic heuristic as an indication of the future load on a ws. The resources that are to be predicted are as stated below:

- CPU time.
- I/O.
- Memory.
- Communication.

The load on a machine is the aggregate resource requirements of each process running on the system. The load index on a particular host is considered as

$$L =< CPU, IO, M, C > \tag{3.1}$$

where

for all i <= n

$CPU = \Sigma cpu_i$

$IO = \Sigma io_i$

$M = \Sigma memory_i$

$C = \Sigma communication_i$

n is the number of running processes.

As the load requirements of a process are based on the history, a job that is running for the first time does not have a history. As the resources cannot be predicted for a new job we have to fall back on some default values for each resource. The default values are taken to be system dependent.

The other load indices used to compute default values are:

- Workstation cpu utilization(cpuu), as percentage.
- I/O rate(ior), as bytes/sec.
- Used Memory(um), as bytes.
- Free Memory(fm), as bytes.

- Communication rate(cr). as bytes/sec.
- Average queue length(aql). as number of processes.

The default values (DEF_CPU,DEF_IO,DEF_MEM,DEF_COMM) are calculated from the above load indices and are given below:

DEF_CPU = (cpuu * t)/aql

DEF_IO = (ior * t)/aql

DEF_MEM = um /aql

DEF_COMM = (cr * t)/aql

t is a system dependent time interval.

Load monitoring policy

The load metrics that constitute the load value of a particular host have been considered in the previous discussion. A hybrid load monitoring policy is used which is a combination of a distributed and a centralized policy.

In a distributed policy every host on the DCS maintains a global database which contains the load information from each host on the network. Every host sends its load information to every other host periodically. In a centralized policy a single host is selected as a database location. Every host sends its current load information to the central database.

In a hybrid policy a centralized host known as a Load Information Center(LIC)

collects the load information from all the hosts and forms a load vector. This load vector is sent to all hosts so that each host has the global load information.

Individual hosts collect the load periodically and feed it to the LIC. As discussed previously the load values for a new job that has no history needs some default values. To calculate the default values the LM collects the following information periodically.

cpuu = (ws_uptime - cpu_idletime) / ws_uptime.

ior = ws_io/ws_uptime.

cr = ws_comm/ws_uptime.

um and fm.

The interval used for the periodic invocation of the LM is variable and can be changed as per the system requirements. The load information collected periodically is sent to the LIC in the format as shown in Figure 3.1.

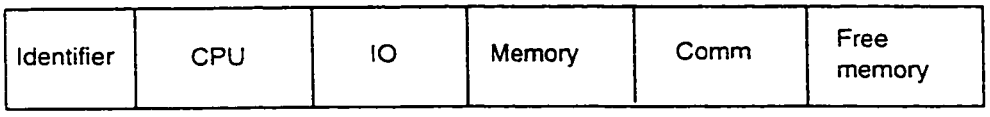


Figure 3.1: Local Load Format

The LIC collects the load information from each live machine, which is formatted as a global load vector and sent back. The format is depicted in Figure 3.2. Each ws over the network has a LM running. The LM at the LIC has extra functionality of collecting the load information and formatting the global load vector. The functionality of a LM is summarized in Algorithm 3.1.

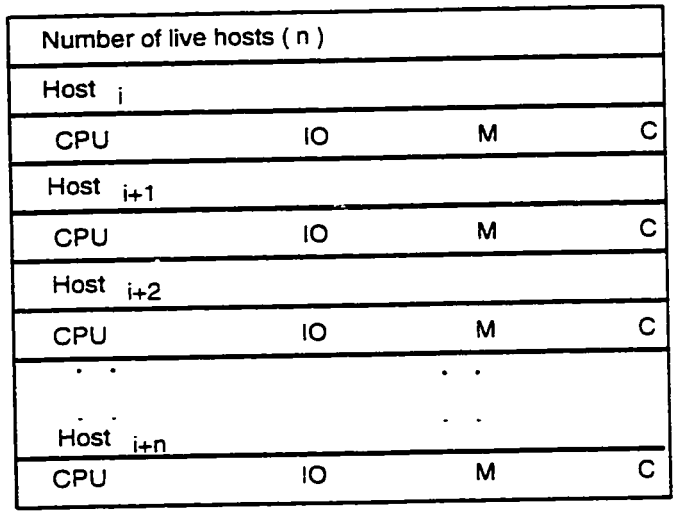


Figure 3.2: Global Load Vector

3.2.2 Application Agent

The functionality of an AA is to provide an interface between the application and the LBFTS. A user submits an application to the AA. The AA in turn uses the user

Algorithm : *loadMonitor*

```
{
    Get the number of live hosts.
    For( all hosts on the Network )
        Collect the system load indices:
        Calculate the default values:
        Find the aggregate future resource requirements of all processes:
    if(host not= LIC)
        Send the collected load information to LIC:
        Receive the formatted Load Vector from LIC;
        Store the Load Vector in a load database;
    if(host = LIC)
        Receive the load information from each host:
        Format the Load Vector:
        Send the Load Vector to every live host:
}
```

Algorithm 3.1: The *Load Monitor*

interface library functions, which are to be discussed, to communicate with the LB and the FTS.

An AA executes till the application is running. Once the application ends, the AA extracts the resources used by the application and sends the information to the LBS. Each application needs to have an AA as depicted in Figure 3.3.

The functionality of AA can be summarized as in Algorithm 3.2.

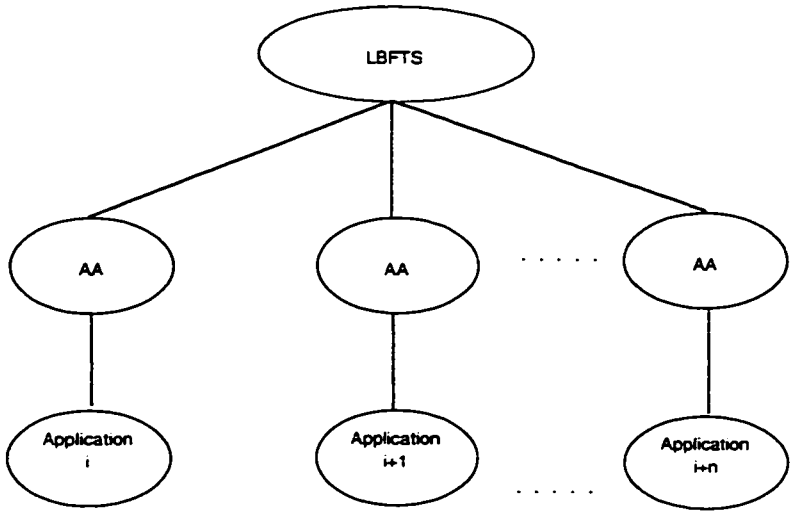


Figure 3.3: Application Agent

Algorithm : *AA*

```
{
    Read the application name and its current working directory.
    Send the above information to the LBS and FTS;
    Inform the user of the location of execution:
    if(local execution)
        wait for the application termination:
        if(application exits)
            Read the cpu,i/o,memory and communication resources used:
            Send the above information to the local LB;
    else
        End.
}
```

Algorithm 3.2: The *Application Agent*

3.2.3 Application Identifier

An application has a name given by the user. In a multi-sharing environment there can be many users and the naming mechanism used by different users may be the same. This may lead to conflicts in application identification.

The main aim of an AI is to uniquely identify the applications and predict the resource requirements of an application. Different cases that can be encountered in application identification are as stated below:

- Different applications may have the same names.
- The same application may have different names.
- Different applications may have different names.
- The same applications may have the same names.

From the above, we can notice that application identification should be independent of names. The method used is to find the checksum of the application binary and use it as its identifier. This is unique as two different applications that have the same name will have different checksums and the same applications with different names will have the same checksum, thus satisfying all the conditions listed above.

The application identifier is invoked by the load balancer whenever an application enters or leaves the system.

3.2.4 Load Balancer

LB is the brain of the Load Balancing Subsystem. The most important issue to be considered in the design of a Load Balancer is to define a proper objective function. The objective function can have three alternatives as stated below:

- Balanced resource utilization.
- Minimization of response time.
- Minimization of completion time.

In Balanced resource utilization the basic idea is to minimize the idle times of the workstations. The Minimization of response time is to execute applications on idle machines if the local machine is heavily utilized. In this case we have to consider whether application relocation will reduce the overall completion time. If the time taken for relocating the application and getting back the results of the execution to the initiating site is more than local execution time then the purpose is defeated. The third alternative is the same as the second one.

The heuristic used is based on good initial placement of tasks. If there is an imbalance later. then some tasks are relocated to some other less loaded hosts. The LB considers the following applications for load balancing

- Independent applications.
- Communicating applications.

An independent application consists of a single task. A communicating application is a group of tasks involved in communication with one or more tasks. The heuristic used considers the following inputs:

1. The cpu.i/o.memory and communication requirement of a task.
2. A task graph in the case of a communicating application.
3. Current load on the workstation.
4. The number of live hosts.
5. The priorities of the workstations if there are any.

The cpu. i/o. memory and communication requirements of the task are predicted based on the history. The current load on each host is supplied by the LM. Information about the number of live hosts is provided by FTS. Other inputs are accessed by the LB from different configuration files.

Some of the issues that arise in the design of a Load Balancer are

- Finding whether the system is suffering from a load imbalance.
- Finding the workstations that are heavily loaded and the ones that are lightly loaded.
- Determining the suitable workstations to execute the local jobs remotely.
- Finding the jobs that are suitable for load balancing.

- Choosing the right load balancing policy.

Using the load value of each workstation determining whether a particular ws is heavily loaded or lightly loaded is just a comparison of the load values of each ws.

Four types of tasks are handled by the LB as listed below.

1. Short.
2. Cpu bound.
3. I/O bound.
4. Communicating.

Short tasks are decided to be executed on the ws they have arrived at. The other three types of jobs are considered for load balancing. Algorithm 3.3 summarizes Jlb and Algorithm 3.4 summarizes the periodic load balancing.

Algorithm : *Jlb*
Input : predicted cpu requirement (*pcpu*)
Input : predicted io (*pio*)
Input : predicted memory (*pmem*)
Input : predicted communication (*pcomm*)
Input : predicted completion time (*pcomp*)
Input : global load vector(*LV*)
Output : destination (*d*)
{
    if(pcpu < shortproc_Time and pcomp < shortproc_Time)
        d = local ws:
    for every ws_i, i = 1.n
        if(pmem <= fm_i) ws_i is a suitable candidate:
    sort the LV according to CPU.IO. M and C load:
    if(cpu bound independent application)
        d = light CPU ws:
    elseif(io bound independent application)
        d = light IO ws:
    elseif(communicating application)
        get the task graph:
        for every task in the application
            if(task communicates with previous task in the graph)
              if (d of previous task is lightly loaded) d = d of previous task:
            else
              d = light C ws:
}

Algorithm 3.3: Job Entry *Load Balancer*

Algorithm : *Plb*
Input : global load vector(*LV*)
Input : task stay time (*TS_TIME*)
Input : predicted task completion time (*PTC_TIME*)
Input : checkpoint interval (*CP_TIME*)
Output : Job List (*JL*)
{

```
    Sort the LV according to CPU_IO_M_C;
    for every independent task
        if(TS_TIME > (PTC_TIME - ε) and TS_TIME > CP_TIME)
            d = light CPU ws;
        if(suitable jobs are available) return JL;
    for(every communicating task)
        if(TS_TIME > (PTC_TIME - ε) and TS_TIME > CP_TIME
        and not Relocated task)
            d = light C ws;
        if(suitable jobs are available) return JL;
        else No load balancing;
```

}

Algorithm 3.4: Periodic *Load Balancer*

The LB policy is distributed and is based on the hybrid load monitor as discussed in the previous section.Every ws has global load information provided by the LM. Based on the combined load vector of all workstations, every workstation implements its own local Load Balancer. The task of each Load Balancer is to do the Job entry and periodic load balancing for that particular ws. LM keeps track of the load values on each workstation. This information is used by the Load Balancer. The local load database and the global load vector database is shared by both the systems as depicted in the Figure 3.4.

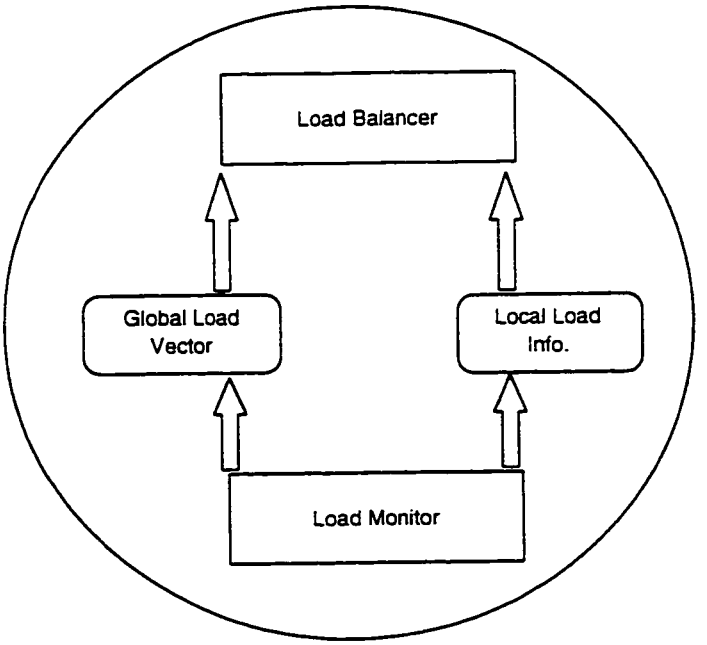


Figure 3.4: Sharing of load databases

3.2.5 History Manager

One of the crucial components that is utilized by the LBS is the History management mechanism. Prediction of application resource requirements is fully based on the history.

The issues involved in designing a History Manager are

1. The location of the history database.
2. The resources that are to be recorded.
3. Database operations.

Location: The location policy is dependent on the design of the Load Monitoring policy. The LIC is responsible for managing the central database which is then passed onto the individual hosts. From the above we can notice that a hybrid location policy is used for managing the history database.

Format of the database: The resources that are to be recorded in the history database are

1. CPU, Memory,I/O and Communication of each task.
2. Task completion time.
3. Task Id.
4. Task termination timestamp.

Task termination timestamp is recorded as the design requires to keep track of the age of records in the database and if the age of a particular record exceeds some pre-defined limit then that record is deleted. Such type of records may not prove useful in predicting. especially the system load. The format of each record in the database can be depicted as in Figure 3.5.

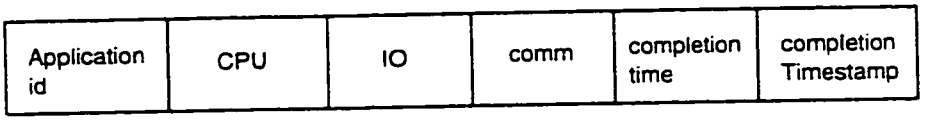


Figure 3.5: History Database Format

Database Operations: The operations to be performed are insertion. deletion and search.

Insertion of records is done at the end of the database. Deletion of records is done whenever the record ages. Search is the most important operation which is done frequently by the load balancing system. The database is searched when there are application entries and exits. The primary key of the database is the task id. The functionality of a HM is summarized in Algorithm 3.5.

```
Algorithm : HM
{
    if(application entry)
        Get application ID from AI:
        Search the history database:
        if(history available) return the record to LB:
    if(application exits)
        Get the resource usage from AA:
    if(ws = LIC)
            Update the history database:
        else
            Send the resource usage information to LIC:
}
```

Algorithm 3.5: The *History Manager*

3.3 Design of FTS

The design of the two components of FTS, the interface with other components and the algorithms used in the design are discussed in this section.

3.3.1 Fault Detection Mechanism

The first step is detection of machine failures. The FDM provides the failure information regarding the group of active workstations to FTS and hence to LBS. The mechanism assumes a virtual ring of workstations. The workstations over the network are connected logically in a ring.

Each ws is assigned a static logical number. The number is static in the sense that it does not change as the workstations come up and go down. Every ws in the virtual ring has a left and a right neighbor. Failure detection of a particular ws is the responsibility of its left neighbor. Each ws sends a "ARE YOU UP" message to its right neighbor and expects a "ALIVE" message as a reply, to consider it alive. The above is depicted as in Figure 3.6.

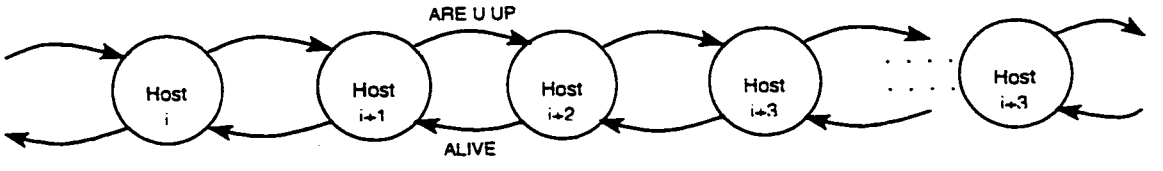


Figure 3.6: Fault Detection

The protocol used to detect failures is

1. Role of left host.

   - Send "ARE YOU UP" to right neighbor.
   - Wait for a reply for a duration of period.
   - If no reply, repeat "ARE YOU UP" message.
   - If no reply broadcast a ws failure message.
   - If a reply, the right host is considered alive till the next period.

2. Role of right host.

   - Receive a "ARE YOU UP" message.
   - Send a "ALIVE" message.

3.3.2 Fault Recovery Mechanism

The second step involved in FTS is the fault recovery. The responsibilities of FRM are:

- Virtual ring management.
- Assigning backup roles.
- Requesting LBS to restart/restore the failed applications.

Virtual ring management

The virtual ring changes dynamically as the workstations become alive or dead. There should be a mechanism to reconstruct the ring for every change as shown in Figure 3.7. Each host keeps track of its neighbor and keeps changing its neighbor as the hosts become alive or dead. From the figure we can notice that at a particular instant A has its left neighbor as X and right neighbor as B. A failure at ws B results in the change of right neighbor of A from B to C.

Backup Roles

Recovery can be achieved by keeping intermediate execution sequences of an application. If an application does not run to completion due to machine failure, then the system will be able to recover the applications by restoring them onto some live ws. The crucial issue is the decision for the backup machine. For a simple and reliable solution the right neighbor is considered the backup machine. If we consider Figure 3.7 the host responsible for detecting failures on host B is host A. When host B goes down host C will become the right neighbor of host A. Hence if we keep the application backups of host B on host C then as host A knows its current right neighbor it has to inform the LBS on host C to restore/restart the applications. Keeping the application backups on a right neighbor is easy as the host does not have to keep track of its left neighbor also. Hence host C is more suitable for backup than host A which is the left neighbor. This is depicted in Figure 3.8. Backup of an application

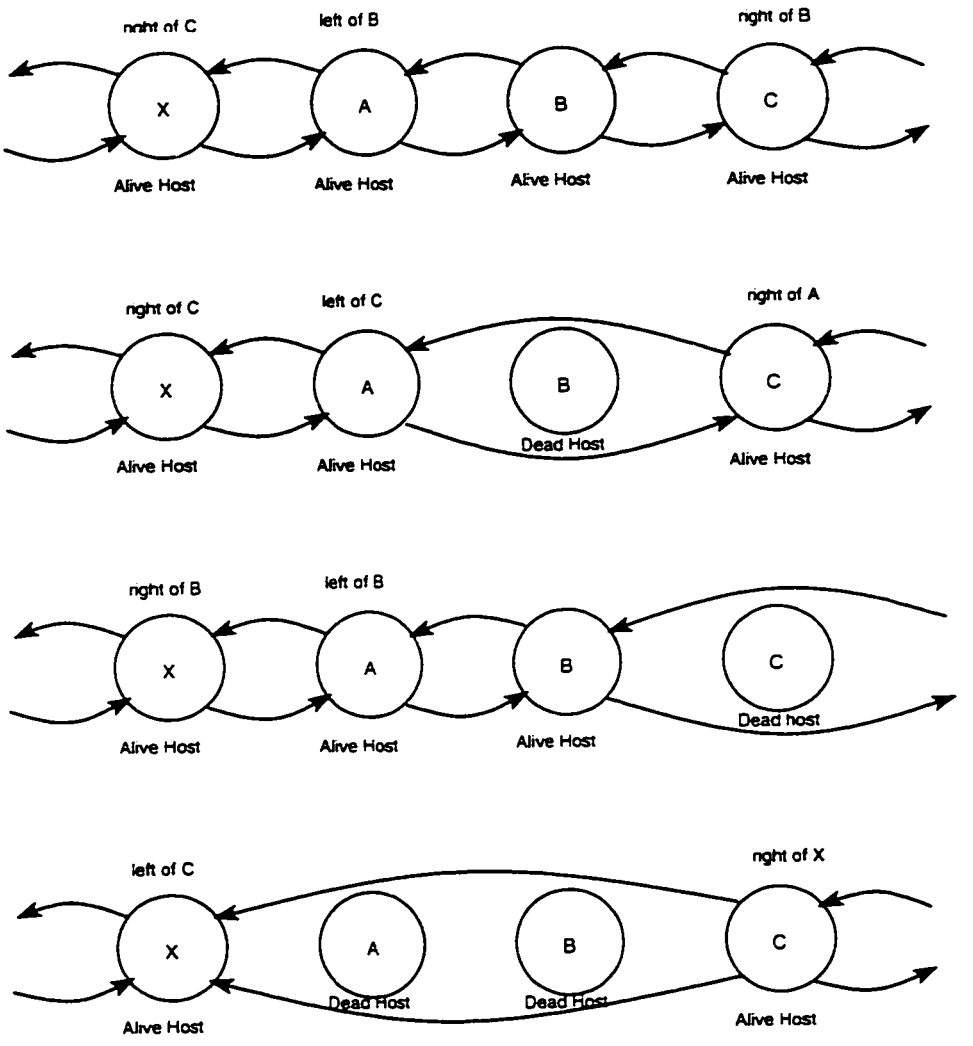


Figure 3.7: Ring Management

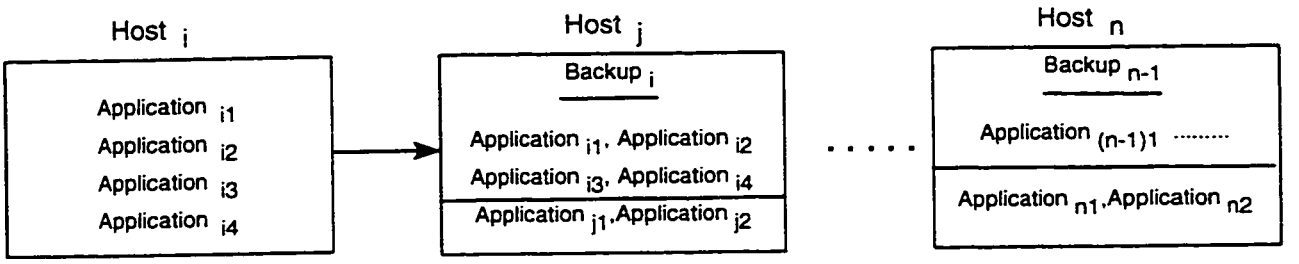


Figure 3.8: Backup Management

is to provide roll back in case the application fails. The roll back method is based on checkpoint facility. In order to roll back we need to have the most recent copy of the checkpoint. Every application that enters the system gets registered with the PMS and these applications are checkpointed at a system specified interval.

Application restore/restart

The final step involved is restoring the failed application. Whenever FDM detects failures in its right neighbor it informs the LB on the backup host to restore/restart the application. The LB reallocates the applications in a load balanced manner.

The functionality of FDM and FRM can be summarized as in Algorithm 3.6.

3.4 Component Interaction

This subsection deals with the protocols of interaction among the various entities of LBFTS. Any distributed algorithm is based on communication of messages between its tasks. Since the tasks are distributed and somewhat centralized it is necessary

Algorithm : *FTS*

```
{
    Get the host name and number of hosts over the network:
    Send a broadcast message at bootup:
    Determine the static right neighbor:
    Send a up message to the right neighbor periodically:
    if(reply message)
        neighbor is alive:
    else
        send another message and wait for a reply:
        if(reply)
            neighbor is alive:
        else
            neighbor is dead:
            broadcast a ws down message:
            change the right neighbor:
            send a restore message to the LB on the new neighbor;
}
```

Algorithm 3.6: The *FTS*

that a receiver expects what a sender has sent and takes necessary action based the contents of the received message. A protocol can be defined as a set of rules followed by the communicating tasks to complete their interaction. The details of all messages and their semantics are presented in chapter 4.

3.4.1 AA Protocol

The AA protocol is responsible for handling the communication between the applications and the LB, the FTS and PMS. Application entry is announced to all the above systems. The entry messages are required by the LB to subject the application to load balancing, required by the FTS to record whereabouts of all the applications running on the system, required by the PMS to register the applications for checkpointing and migration. Application completion messages are required by the LB to record the history, required by the FTS to remove the application record from the list of currently running applications. The protocol can be depicted as in Figure 3.9.

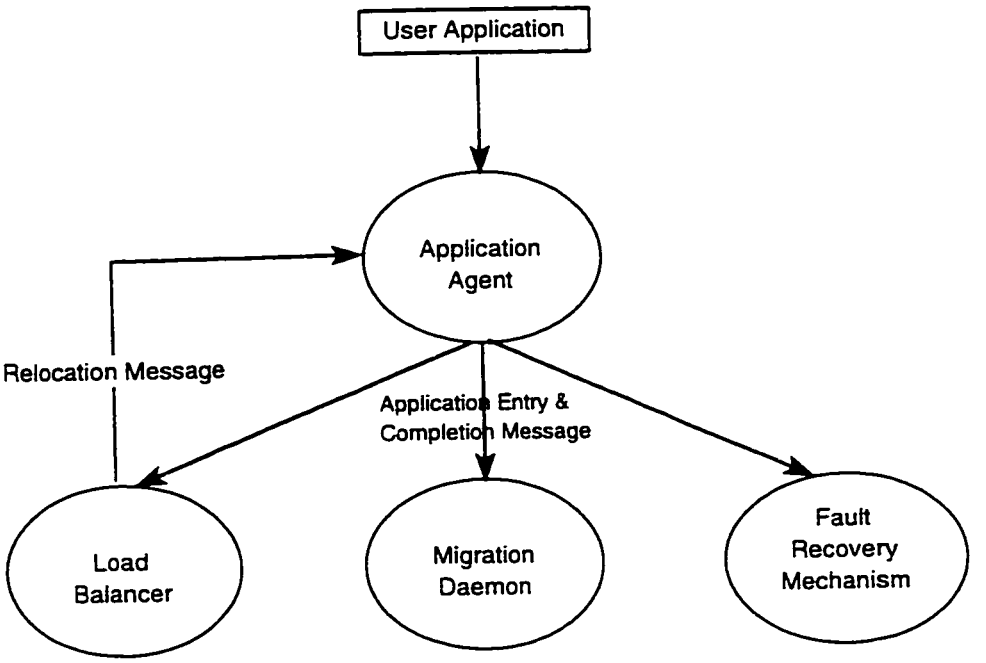


Figure 3.9: Application Agent Protocol

3.4.2 LBS Protocols

LM protocol

The protocol is responsible for handling communication between the local Load Monitors and the LM on the LIC and vice versa. The load information is communicated to the LIC by LM's on all workstations and the LIC combines the load information into a load vector and sends it to the LM's on all the workstations in the form of a message.

LB protocol

The protocol is responsible for handling the communication between the distributed load balancer. The protocol handles the application entry messages, application termination messages, application relocation messages, application restore messages, relocated application completion messages and history update messages. The two protocols can be depicted as in Figure 3.10.

3.4.3 Fault Tolerance Protocol

The protocol handles the messages between the distributed FTS daemons. It handles "ARE YOU UP" and "ALIVE" messages. The protocol can be depicted as in Figure 3.6.

3.5 Design Alternatives

The design alternatives for LBS have been discussed in the literature review. The design of FTS may have different alternatives. The alternatives may come into picture while designing the FDM and the backup machine selection. An alternative for the fault detection is a virtual ring where every ws checks for its left neighbor

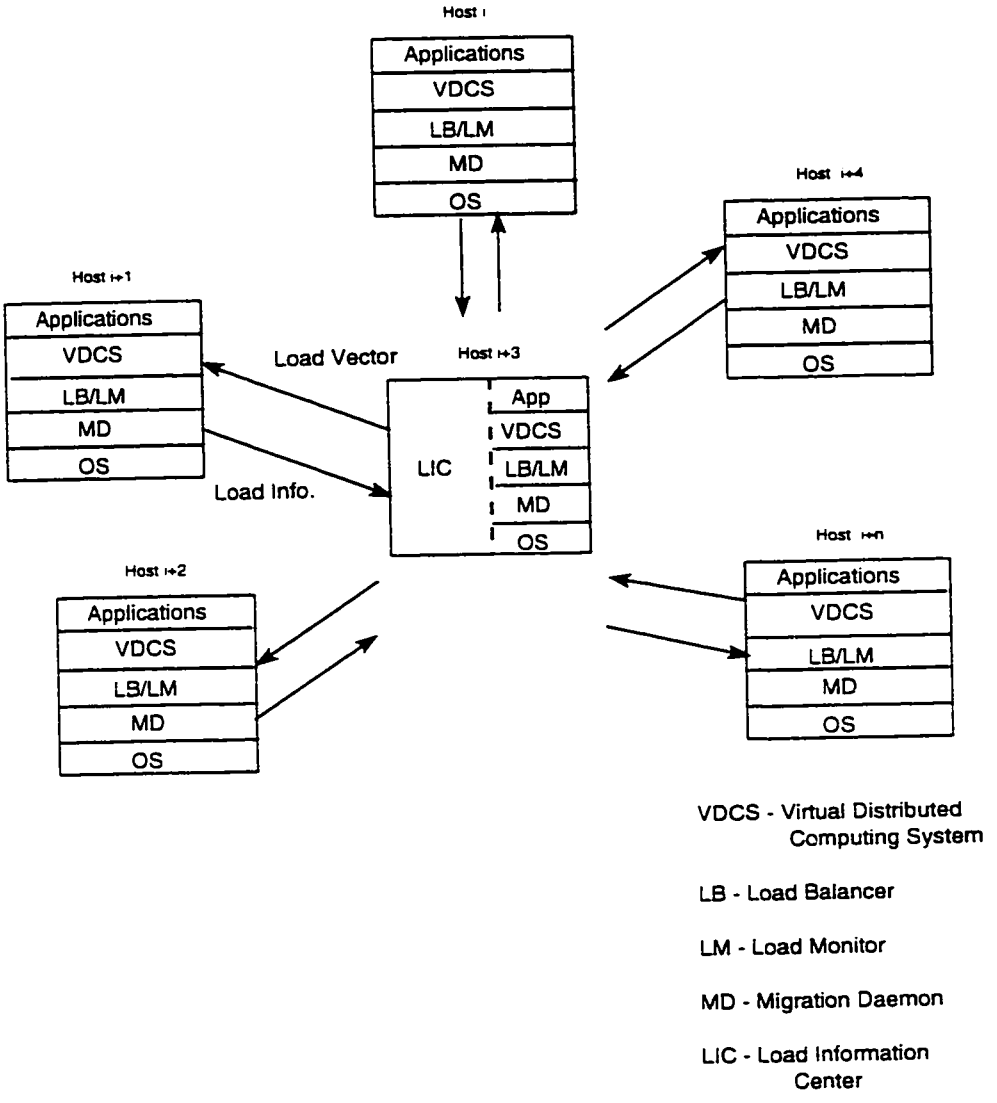


Figure 3.10: LBS Protocol

and right neighbor as shown in Figure 3.11. The fault detection process will speed

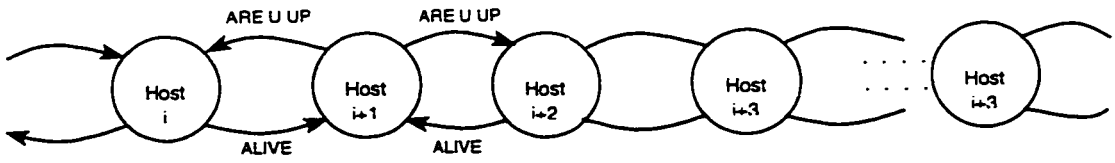


Figure 3.11: Double Virtual Ring

up but at the cost of increased number of messages and hence increased processing overhead. Based on the above mechanism deciding who is responsible for keeping track of the checkpoints of applications is another issue. An alternative may be to keep the checkpoints on the right neighbor or the left neighbor. The design becomes more complicated as there has to be a maintenance mechanism for two virtual rings instead of one virtual ring as in this system. Each workstation has to keep track of two neighbors and send messages to two neighbors.

Chapter 4

LBFTS Implementation

This chapter gives a detailed report of the implementation of LBFTS based on the design discussed in chapter 3. LBFTS is implemented as a set of distributed daemons running over the network. Each daemon works on its own with minimal message exchanges with other daemons.

The interface with PMS and the usage of services provided by the PMS are discussed. The overhead involved due to the different components of LBFTS and the overhead due to PMS is analyzed.

4.1 The working environment - LINUX

The working environment chosen for the implementation of LBFTS is a distributed system comprising of a network of PC's running the LINUX operating system. Let

us look into the features provided by LINUX and the reason for choosing it as the implementation platform.

4.1.1 Features of LINUX

Linux is a clone of the Unix operating system that runs on IBM PC compatible machines with *Intel-386/486/Pentium* or equivalent processors. It was written from scratch by Linux Torvalds (a graduate student at the Helsinki University of Technology, Finland), in collaboration with an enthusiastic, world-wide group of volunteers interacting through the Internet. Linux is a growing OS with lots of scope for incorporation of new provisions.

Linux is a full-fledged operating system that provides all the capabilities normally associated with commercial Un*x systems. It supports most of the features found in other implementations of Unix, plus quite a few that aren't found elsewhere. It is a complete multitasking, multiuser operating system and is mostly compatible with a number of Unix standards at the source level, including IEEE POSIX.1, System V, and BSD. All its source code, including the kernel, device drivers, libraries, user programs, and development tools are freely distributable. The major features of Linux can be summarized as follows:

- Support for pseudo-terminals, loadable keyboard drivers, and virtual consoles.
- POSIX compatible job control.

- Kernel emulation for 387 FPU instructions.
- Support for a variety of filesystems such as ext2fs, Minix, Xenix, MSDOS and ISO 9660 CD-ROM filesystem.
- Complete implementation of TCP/IP networking; including SLIP, PLIP and PPP.
- Support for NFS, NIS, FTP, Telnet, NNTP and SMTP.
- Support for demand-paged loaded executables, disk paging and dynamically linked shared libraries.
- Excellent compilers for C, C++, Pascal, Modula-2, Oberon, Smalltalk and Fortran, standard utilities for text/word processing and the X-Window system
- Password security, file protection, multiple logins, virtual memory and multi-tasking.
- Provides workstation capabilities on top of inexpensive PCs.

The obvious reason for choosing LINUX as the implementation platform is that PMS has been implemented in LINUX. The reason PMS was implemented in LINUX was the free availability of linux source code. Its free availability and the fact that it runs on PC's has made it a very popular OS in a span of just a couple of years. It presently enjoys a wide user base in both academic and business environments.

Utilizing the enhanced version of linux to implement LBFTS would make Linux a full fledged distributed operating system.

4.1.2 Process Migration System

Let us discuss PMS briefly. PMS comprises of three components : the E-kernel, the PMS library and a set of distributed Migration Daemons(MD). The functionality of migration is handled by the Enhanced kernel(E-kernel). The E-kernel comprises of implementation of two new system calls sysCheckpoint( ) and sysRestore( ). sysCheckpoint( ) provides a way to checkpoint applications. sysRestore( ) provides a way to restore checkpointed applications. The PMS library provides a set of routines that form the interface to the two system calls. The MDs play a major role in checkpointing and migration, apart from ensuring sustained communication among the migrating processes belonging to distributed applications. The PMS can be depicted as in Figure 4.1.

Checkpointing forms the fundamental mechanism needed to provide migration capability. This mechanism is implemented within the kernel. Restoring forms the fundamental mechanism to reinstate the process image from where it was stopped in case of machine failures.

The PMS library routines are implemented outside the kernel. The reason for this is to maintain easy portability. The library routines play a major role for providing support for migration of processes by interacting with the MDs. In

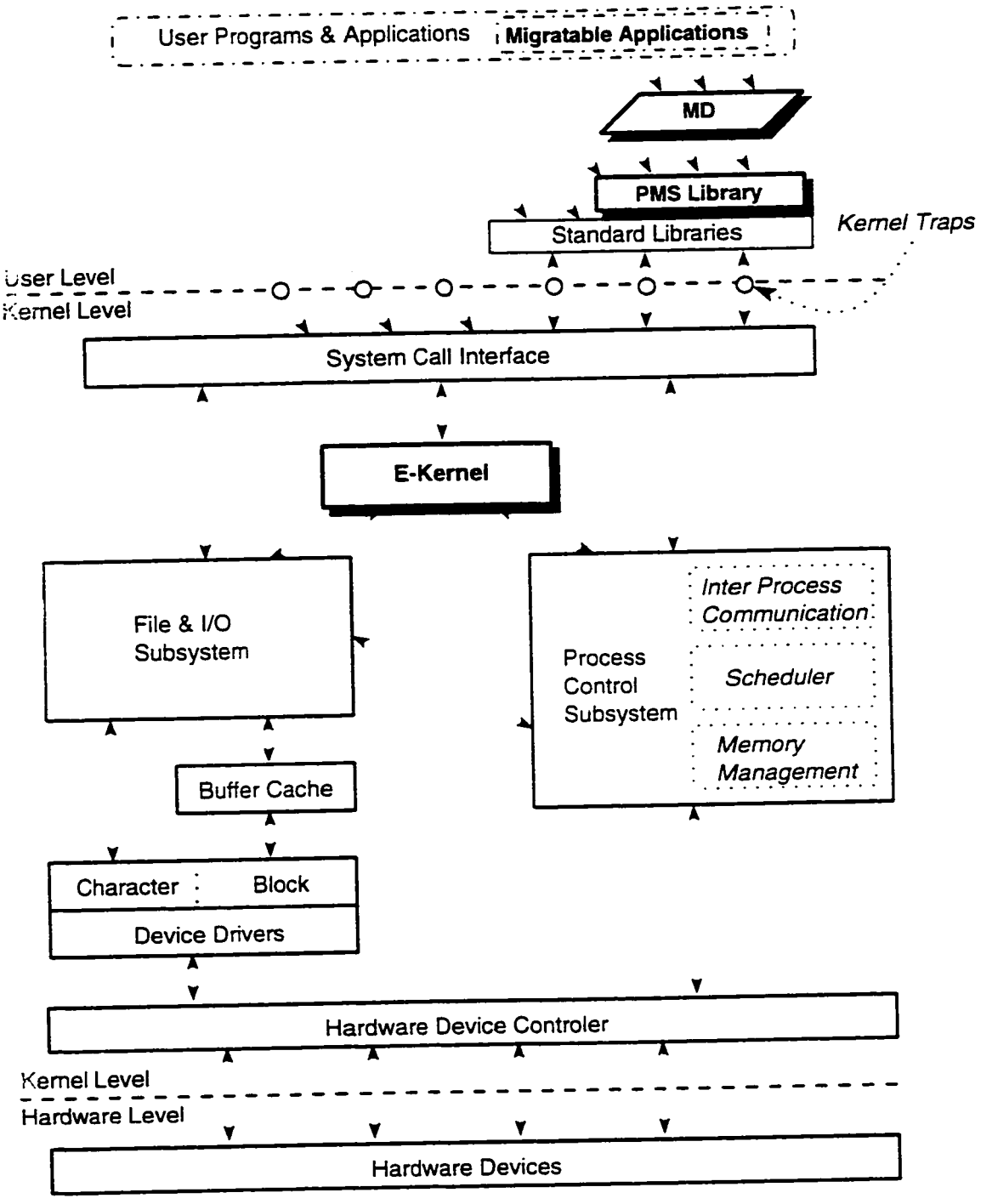


Figure 4.1: PMS

order to be migratable, an application has to be registered with the local MD. The system provides different types of registrations. The type we are interested is the regCheckpoint which checkpoints an application after every checkpoint interval.

The PMS has been discussed briefly. The services provided by this system are used for periodic load balancing and for providing a fail proof environment.

4.1.3 LBFTS platform

A network comprising of i486 processors is the basic setup used. The PC's are connected through a 10 Mbps ethernet cable. Each machine has the Enhanced version of LINUX and runs four daemons: LM daemon, LB daemon, FTS daemon and MD.

The implementation details will be presented in the next section. These daemons are started at the system bootup and since then, they will be running except when a machine failure occurs.

The system developed does not assume a global file system. Hence the required functionalities that are provided by a global file system are implemented within LBFTS.

4.2 Implementation details

In this section the implementation details for the design discussed in the previous chapter are reported. The issues that have not been mentioned explicitly are listed and discussed.

4.2.1 Load Monitor Daemon

Load Monitoring forms the basic mechanism to provide load balancing capability. As discussed in the previous chapter determination of load metrics and the load monitoring policy had been the main concerns in the design process. According to the monitoring policy each ws monitors its own work load and sends the information to the centralized LIC. The LIC distributes the combined load vector to other workstations. The program that monitors the work load on a particular ws is known as Load Monitor daemon.

The message handling and interpretation is one of the main parts of a Load Monitor. Each ws sends the load information periodically. The period is a tunable parameter which depends on the real traffic over the network. The time interval chosen is 2 seconds, so each ws has to collect its load information every 2 seconds and format the message and send it to the LIC.

The LM depends on the message handling provided by the TCP/IP networking. Different types of messages are handled by LM. The message listing is as below,

- Local load messages sent to LIC.
- Global Load Vector update messages sent from LIC.

Each local LM sends the load information to the LM, at the LIC, in the form of a message. The message is to be sent in a meaningful way so that the receiver of the message should be able to extract the information from the message. The format of the message sent by a local LM to LIC is depicted in Figure 4.2.

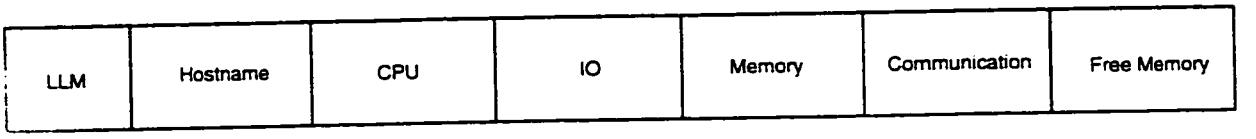


Figure 4.2: Load Update message

The header field "LLM" identifies the message as a load update message from a ws. The second field identifies the name of the ws from which the load update has been sent. The remaining fields identify the load information of the sending ws.

Each local LM receives the global load vector sent by the LM at LIC in the form of a message. The information is recognized by checking the first field of the global load vector. The global load vector has been depicted in Figure 3.2. It is prefixed with a field called LIC for message identification.

The implementation details required to implement a load monitoring policy have been outlined. A LM uses some configuration files to perform its functionality. The configuration files along with their contents and usage are explained below:

- Update_Time: This parameter specifies the periodic interval for load update.This information is extracted by the LM from a configuration file "Parameters".

- LIC_Name: The information is used to know the hostname of the centralized LIC. This information is extracted by the LM from a file "lic.conf". This configuration file is updated by the FTS.

The information provided by the configuration files is crucial for the working of the LM. The LM is invoked periodically for load collection and update. An alarm is set which invokes the LM to collect the system load indices as discussed in chapter 3. The system load indices are read from **proc** file system. The information extracted from the proc file system and the source of information is listed in Table 4.1.

Information	Source
Cpu Utilization	/proc/uptime
IO rate	/proc/stat
Used, Free Memory	/proc/meminfo
Communication rate	/proc/net/dev/net
Load average	/proc/loadavg

Table 4.1: The proc Filesystem

The information is processed to get the default values discussed in the previous chapter. The load collection is followed by the loadupdate message. The steps that are followed are:

- Extract the name of the LIC.

- Format and send the message to LIC.

The LM handles the global Load Vector sent by the LIC. When a LM receives a load vector it has to decode the information and update its local load vector file, to be used by the LB.

4.2.2 Load Balancer Daemon

The design of a LB has been discussed in the previous chapter. The implementation details will be discussed in this section.

LB has to do Job entry load balancing and Periodic load balancing. Let us first discuss the message handling done by the load balancer. The types of messages handled by the load balancer are:

- Local process entry.
- Local process completion.
- Process relocation.
- Relocated process completion.
- History update.

Local process entry: Whenever an application enters the system the application entry is to be recorded by the system for load balancing and history management purposes. The message format is depicted in Figure 4.3. The message identification

is done by using the first field P_ENTRY of the message.

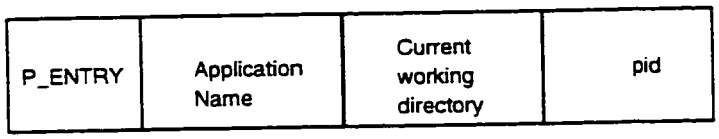


Figure 4.3: Local Process Entry message

Local process completion: A message is sent by the application agent whenever the application terminates. The information sent in this message is used by the Load Balancer and the History Manager. The format of the message is depicted in Figure 4.4.

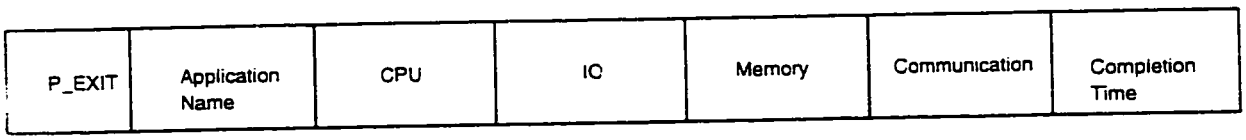


Figure 4.4: Local Process Exit message

The first field P_EXIT identifies the message. The application name is used by the load balancer to delete the application entry from its list of currently applications. The remaining fields are used by the history manager to update the history.

Process relocation: One of the capabilities of the load balancer is initial place-

ment of applications. An initial placement can be either the local ws or a remote ws based on the load situation. If a particular application is remotely placed then the local LB has to inform the remote load balancer of the relocation. The format of a relocation message is depicted in Figure 4.5.

REL	Application name	Current Working directory	predicted CPU	predicted IO	predicted memory	predicted Comm	predicted completion time

Figure 4.5: Process Relocation message

The first field REL identifies the message. The second and third fields are used by the load balancer for assigning the application to an application agent to execute the application. The fields predicted CPU, predicted I/O, predicted memory and predicted comm and predicted completion time are the resource requirements predicted by the load balancer on the machine the application was actually initiated. This information is sent in the message to avoid re-estimation of resource requirements by the remote LB, thus improving the efficiency.

Relocated process completion: An application that has been relocated by job entry LB or periodic LB has to send its resource usage information to the application initiator. The LB is responsible for receiving this information and sending the information to the central history database. The format of the message is depicted

in Figure 4.6. The first field REM identifies the message as a remote job completion message.

REM	Application name	Current Working directory	CPU	IO	memory	Comm	completion time

Figure 4.6: Relocated job completion

LB receives messages from the FTS for restoring failed applications. The messages are listed below:

1. Process restoration.
2. Left host process entry.
3. Proxy host process completion.

Process restoration: LBFTS provides a fail safe environment using the FTS. The FTS keeps track of the machines and a machine failure is detected and the LB is informed of the same. The responsible LB should take care of the applications that were running on the failed machine. The message sent by the FTS for application restore is a Process Restoration message. The message contains a single field RESTORE which identifies the message. The LB takes the following steps when this message arrives:

- Get the list of applications that are to be restored/restarted.
- Consider each application as a new process and perform the load balancing.

Left host Process Entry: The process restoration messages are used by the LB to handle the applications on the left host. The LB should have a list of all applications that are running on the left host as a failure on that machine is followed by a recovery on this machine. An application entry on a proxy host is followed by a message to the right host. This message is sent by the FTS. The format of the message can be depicted in Figure 4.7.

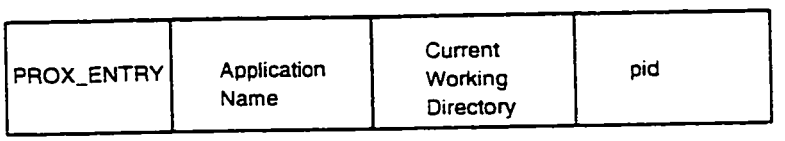


Figure 4.7: Proxyhost Entry message

The field PROX_ENTRY identifies the message.The LB records the application for future use. A left host is called a proxy host.

Left host process completion: The reason for this message follows from the discussion of the above message. An application exit on a proxy host should be get to known to delete the entry from the list of applications currently running on the proxy host. The format of the message is depicted in Figure 4.8. The field PROX_EXIT identifies the message.

Figure 4.8: Left host exit message

Let us now discuss some of the tunable parameters that are used by the LB. The parameters are read from a configuration file 'Parameters'. They are listed as below.

- Periodic_Time: The time used by the daemon for periodic invocation of the LB. The range of values for this variable is {15.30.60} seconds.

- Shortproc_Time: The value that tells the LB what a short process is. The possible range is {5. 10. 15} seconds.

- Cpudiff: A tunable parameter for load balancing decisions. The LB compares the CPU loads on all machines and determines the lightly loaded host. A host is chosen only if difference between the local CPU load and the remote CPU load is greater than Cpudiff. Cpudiff can have the values {100.300.500} depending on the type of applications.

- Iodiff: When IO comparison is done this parameter is used as explained above. This parameter can lie in the range {5000.10000.20000} bytes.

- Commdiff: When communication based comparison is done this parameter is

used. The difference between the ws loads can be in the range {1000.5000.10000} bytes.

- Cpubound: As the LB chooses different destinations based on CPU. IO and COMM. the LB should decide on a particular destination. The method used is to check whether a particular job is cpu bound or not. If it is cpu bound then the destination is based on CPU. Cpubound can have values in the range {50.100} seconds and above.
- Iobound: It can have the values in the range {50000.100000} bytes and above.
- Commbound: It can have the set of values {5000.7000} bytes and above.

Choosing a suitable application for migration is another issue. The LB daemon keeps track of the applications that are running on the machine in two lists. One list is for independent applications and another for communicating applications. As the migration of communicating applications is given less priority over the migration of independent applications, keeping two lists helps in this motive. Whenever a PLB is invoked it checks for the feasibility of migration of independent applications. The daemon goes through the list of applications one by one and checks for its feasibility. The method used to avoid unnecessary migration of applications is the following.

- Get the median of the list.

- Check for the feasibility starting from the application at the median.
- Choose the most suitable application based on the load situation.

If the above method does not provide atleast a single application then the LB falls back on the list of communicating applications. The method used for independent applications is applicable to the list of communicating applications.

4.2.3 Fault Tolerance Daemon

The FTS daemon constitutes of the FDM and the FRM. The details for FDM are given first. The FTS daemon mainly does message handling and based on the message contents the required functionality is invoked. The types of messages handled by the FTS daemon are listed below.

- Are u up.
- Alive.
- Broadcast_up.
- Broadcast_down.

Are u up: Each host needs to keep track of its right neighbor. The tracking is done by sending an UP message to the neighbor. An UP message consists of a single field UP which identifies the message.

Alive: When a host receives a 'Are u up' message it has to send a reply immediately

to the sender to inform that it is alive. An ALIVE message consists of a single field REPLY which identifies the message.

Broadcast_up: The FTS provides a flexibility for addition of new hosts to the system. The dead host may become alive and this has to inform every other host. When the system boots up the daemon gets initiated and sends a broadcast message to every other host. The other hosts receive this message and update their ring configuration. A broadcast message has the format as shown in Figure 4.9.

Figure 4.9: BCUP message

Broadcast_down: A host detects the failure on its right neighbor. This is followed by a broadcast message informing every other live host of the failure. On receiving this message each host updates the required configuration files. This message has the same format as Figure 4.9, except the first field which is replaced by Bc_down.

We have discussed the types of messages that are handled by a FT daemon. Now let us discuss the configuration file that are used by a FT daemon. The description is as follows,

host.conf: This file contains the information about the hosts that are connected over the network. and the file format can be depicted in Table 4.2. The UP/DOWN

HOSTNAME	HOSTNUM	UP/DOWN
Controller	0	1
SecondHost	1	1
FirstHost	2	1

Table 4.2: The Host Configuration File

flag is updated by the daemon based on the Bc_up and Bc_down messages . An observant reader might have noticed that a host that comes up can only update its own UP/DOWN flag and it has no information about other hosts. The technique used is to give the responsibility of informing to its responsible neighbor.

Now let us consider the ring management mechanism. Based on the information provided by the host.conf file the virtual ring is formed and the actual right neighbor of a particular host is calculated using the formula given below.

$$rightneighbor = (hostnumber + 1)\%MAXHOSTS \tag{4.1}$$

where MAXHOSTS is the number of hosts connected over the network. Using the above formula if we have 10 hosts, the right neighbor of host 0 becomes 1 and the right neighbor of host 9 is 0 and so on.

Whenever a Broadcast_up message is received by a host it checks whether it is its actual right neighbor. if it is then it changes its right neighbor to this new host. Similarly a Broadcast_down message is sent after a failure detection and the right neighbor is changed to the next live right neighbor which is calculated according to

the following formula.

$$rightneighbor = (current\_right\_neighbor + 1)\%MAXHOSTS \quad (4.2)$$

Details of FDM have been provided in the above discussion. The discussion for FRM follows. As discussed in chapter 3 the application backups are maintained on a hosts right neighbor. Applications running on a host are checkpointed periodically which are stored on the local machine. These checkpointed files are copied periodically to the right neighbor for future use by the migration system.

Whenever a fault is detected the fault daemon sends a restore message to the responsible host for restoring the applications that were running on the failed machine.

4.2.4 Application Interface

The application interface to LBFTS is provided through an application agent as discussed in chapter 3. Each application is submitted to LBFTS by an AA daemon which takes care of the application. AA communicates with the LB and gets back the decision taken by the LB which is informed to the user. The AA uses two library functions to communicate with the LBS and FTS and one library function to read the resource usage.

- sendtoLocalLB().
- sendtoLocalFT().

- readProc().

sendtoLocalLB() formats a message to be sent to the LB whenever an application enters or leaves the system. The format of the message is the same as the Local process entry message and Local process exit message. sendtoLocalFT() formats a message to be sent to the local FT daemon to keep track of the application. The AA waits for a message from the LB after a Process entry message is sent. Based on the message it informs the user whether the application has been executed on the local machine or has been relocated to a remote machine.The messages received from the LB are of two types: Continue and Relocated. Continue message indicates the application is started on the local machine and Relocated message indicates the place of relocation.

The functions sendtoLocalLB() and sendtoLocalFT are a part of the library interface discussed in the next section. readProc() is a function that reads the process resource usages from the proc file system.

4.2.5 LBFTS library

The LBFTS library includes the library functions provided by PMS. The library functions are used by the AA. These functions hide the LBFTS implementation from the user. The functions are provided whose semantics are more related to the user domain of problem solving. These functions generate messages in a suitable format for LBFTS to understand. The set of library routines are implemented in

the file mlib-m04.c. The routines implemented in the library cater for

- Registering the application with LBFTS and PMS.
- Extract the resource usage when the application terminates.
- Send and get the information from the LB and the Migration daemon.

4.2.6 History Management

The Load Index used by LBFTS mainly depends on the prediction of resources. As we can see that the prediction is based on the history managed by LBFTS, history becomes a crucial part of the system. The database is a global file which is used by every host. The history management is done by the LIC. So each host sends a history update message to the LIC whenever an application leaves the local machine. The message received at the LIC has the format as depicted in Figure 4.10.

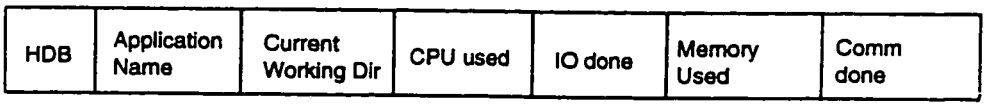


Figure 4.10: History Update message

On receiving this information the history manager checks the history database for a duplicate entry and updates the database with the average values of the new

entry and the old entry.

4.3 Overhead Assessment

This section presents the overhead that occur as a result of the socket interface used by different components of LBFTS and the overhead due to PMS.

4.3.1 Messages

The messages are handled by the socket interface provided by TCP/IP. This is the lowest level of support provided for application programmers intending to code distributed applications by a typical operating system like UNIX(LINUX). This interface has become a standard in contemporary computing. Sockets are analogous to mailboxes and telephones in that they allow users to interface to the network, just as mailboxes allow people to interface with the postal system and the telephones allow access to the telephone system. The position of sockets in the Operating System is depicted in Figure 4.11.

Sockets can be created and destroyed dynamically. Creating a socket returns a file descriptor, which is needed for establishing connection, reading data, writing data and releasing the connection . Having a file descriptor is not enough, extra information which can be considered as an overhead has to be maintained. For a

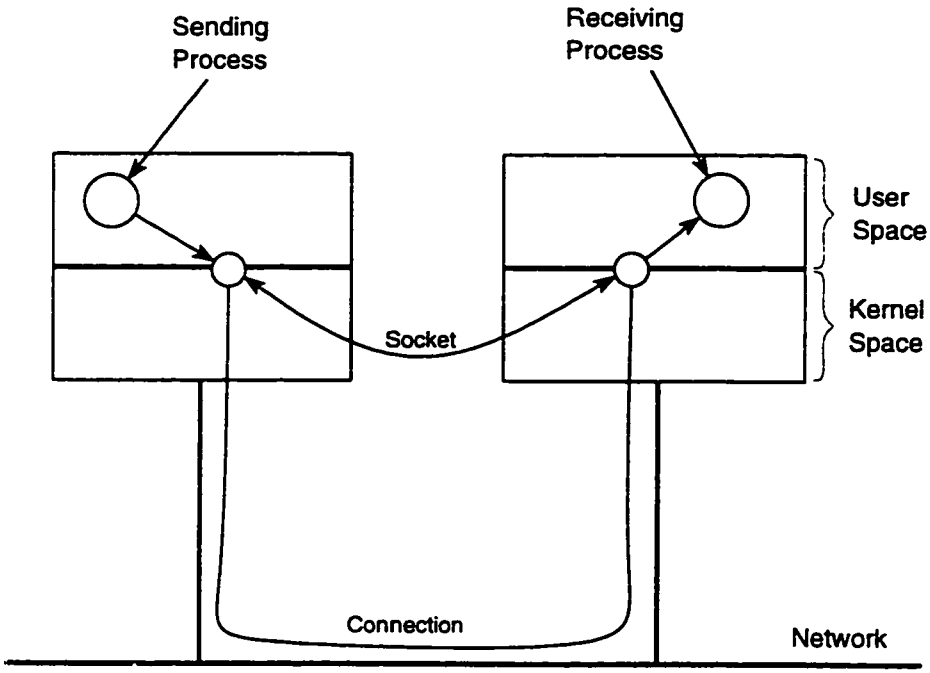


Figure 4.11: Socket Interface

LINUX os it is maintained in the struct sockaddr_in defined in socket.h.

In order to provide transparency and interface to the network, the resultant overheads are unavoidable. These overheads are explained by the types of sockets used. This in turn implies the type of protocol used. The most common protocols are referred to as the TCP/IP suite of protocols which provide the following socket types,

1. Datagram: These type of socket models potentially unreliable, connectionless packet communication.

2. Stream Socket: These model a reliable, connection oriented byte stream.

3. Sequenced packet Socket: These model sequenced, reliable, unduplicated connection based communication that preserves message boundaries.

The type of socket model used by LBFTS is the datagram. The reason behind this is the asynchronous nature of message delivery. The sender sends a message and does not wait for the acknowledgement from the receiver. This avoids waiting time. Similarly messages can be received as and when they arrive. Hence the non blocking nature of the datagram causes less overhead than other types of communication.

4.3.2 Process Migration Subsystem

PMS uses the socket interface for providing inter MD communications and user application-MD communications. Therefore all the overheads associated with the socket interface are valid for PMS. In addition to this other overheads occur because of the following reasons.

1. PMS provides checkpointing of applications. The checkpointing can be considered as the time interval between the instant the checkpoint library function is invoked and the process gets checkpointed. The time depends on whether it is a cpu intensive or io intensive or communication intensive application. The possible overheads are listed in Table 4.3. The maximum overhead that can occur is 1.41 seconds.

Application Class	Checkpoint Time (secs)
Compute Intensive	1.24
I/O Intensive - 1 File	1.17
I/O Intensive - 2 File	1.35
I/O Intensive - 3 File	1.41
Communication Intensive	1.21
Hybrid Application	1.22

Table 4.3: The average checkpoint times

2. Migration: The PMS provides a migration mechanism which consists of messages and an overhead of 4 seconds is needed for restoring the application.

4.3.3 Fault Tolerance Subsystem

The overhead due to messages can be estimated based on the frequency of ARE U UP messages and the frequency of the machine failures. In addition to this overhead each host has to update the host configuration file whenever required. To reduce the overhead due to file updates, the files are memory mapped.

4.3.4 Load Monitor

The overhead due to LM messages has to be accounted for. The frequency of messages depends on the load update interval. For every update interval the network experiences n messages where n-1 are load updates sent to the LIC and 1 message for the Load Vector update from the LIC. The time taken for such messages can be accounted for in micro seconds.

4.3.5 Load Balancer Module

The overheads as mentioned below:

- Application identification.
- Application list management.
- Decision making process.

Application identification is done two times. One at process entry and another at process termination. Resource prediction requires history search. The size of the database adds to the overhead of LB.

The application recording at the LB is done as a linked list which needs search, add and delete operations. The overhead due to the decision taking process shows a considerable effect on the application completion times. The overhead occurs due to Jlb and Plb. Jlb incurs the overhead when applications enter the system. Plb overhead is periodic.

Chapter 5

Experimentation and Results

Performance analysis of LBFTS has to be done in two perspectives: the application perspective and the system perspective. In the application perspective we have to consider the completion time of an application and we have to verify that a particular application executes uninterrupted. From the systems perspective we have to consider the system throughput and the ability to tolerate faults. In our analysis the two perspectives overlap each other. Now let us define the **Completion time** and **Speedup**, which are the metrics used to analyze the system performance.

The Completion time of an application is the clock time which is measured from the application submission to its termination time.

Using the application completion times the Speedup is measured. One is based on the average completion times and the other is based on the completion times. Generally speedup is the ratio of the completion time of an application in sequential

execution(using one machine) to the execution in a load balanced manner(using n machines).

Performance analysis is based on the experiments done using four types of applications as briefly outlined in the next section.

5.1 Experimental Applications

The experiments are done using different types of applications as listed below:

1. Independent CPU intensive.
2. Independent IO intensive.
3. Communication intensive.
4. Real.

5.1.1 Independent CPU intensive

CPU intensive applications are *independent* programs which execute as a single process. Each process will be a stand-alone entity, in that it has no communication/interaction with any other processes currently running on the system and spends most of its time in doing some computation. The experimental setup used is a group of 20 such processes. Each application of this category will be a group of 20 independent cpu intensive processes.

Each process executes a random loop. The UNIX function srand() is used to produce a random loop variable. The group of these processes provide a random environment, as each process's amount of computation is based on a random variable. A representative example is as depicted in Algorithm 5.7. Each process has a different seed value, such that srand produces different random value for different processes.

Algorithm : *CPU*
input: Seed value
{

```
    value = srand(seed);
    loop for value times;
}
```

Algorithm 5.7: A *CPU intensive* program

5.1.2 Independent IO intensive

IO intensive programs are independent processes which do not involve communication among themselves and each process deals mostly with I/O. A heavy percentage of its time is spent in reading data values from one or more files and generating the results in the same/different files.

A representative process in this class is one that reads data from one file and writes the data to another file. The experiment consists of a group of 30 such processes where each process reads and writes the data a random of times. The

random value is similar to the one used in the CPU intensive processes.

5.1.3 Communication intensive

Communication intensive applications belongs to the class of *parallel* applications. This class comprises of two or more tasks which communicate among themselves to produce the desired results. The processes belonging to such applications can therefore be categorized as communication intensive. A representative example of this class comprises of a server and 20 client processes. The clients and the server communicate with each other. The clients communicate with the server a random number of times using the random generator previously discussed. The server sends a reply for each message from the client.

5.1.4 Real application

CPU,IO and Communication intensive hypothetical applications provide a random environment. Analyzing the system using hypothetical applications is useful as the environment can be controlled in terms of application characteristics. A real application like matrix multiplication allows to observe the performance of LBFTS in more realistic conditions. It constitutes of I/O, computation and communication. IO constitutes of reading the input matrices and writing the results to files. Matrix multiplication is compute intensive in the sense that it involves large real multiplication.

The matrix multiplication if not partitioned , is a single process. If partitioned, it comprises of a server and n-client processes. The clients perform multiplication on their own datasets and communicate the results to the server. The server gathers the results and writes them to a file.

If we consider two m x m matrices and n clients, each client computes m/n rows of the resultant matrix. Each workstation is supposed to have the input matrices. From the above, we can conclude that matrix multiplication provides a hybrid environment which involves I/O, CPU and communication.

5.2 Experimental Setup

LINUX is a time sharing operating system. A particular workstation has system processes which share the cpu with the user processes, thus affecting the completion times of the user processes. A user application can be submitted as a single process or can be partitioned into many processes. An application submitted as a single process has to share the cpu with the system processes, whereas a partitioned application has time sharing overhead due to its own processes in addition to the overhead due to system processes.

5.2.1 Setup for Sequential execution

In order to compute the speedup, we must have sequential execution time(completion time of an application). In our experimental setup we have independent(CPU,IO intensive), communication intensive applications and matrix multiplication. Independent applications and communication intensive applications are a group of processes. The system that each application runs on is an ordinary LINUX system. Obviously the effect of time sharing must be considered in the analysis. Based on whether timesharing is considered in the analysis or not, we have two cases for sequential execution:

- Timeshared_Singlemachine execution (Ts_Sm).
- Sequential execution (Se).

In the first case timesharing is considered for analysis whereas in the second case the analysis is done eliminating the time sharing effect.

5.2.2 Setup for LBS

Different cases have been used to analyze the system.

- Local execution(Le).
- Initial placement without migration and checkpointing(Ip).
- Initial placement with checkpointing and without migration(Ipcp).

- Initial placement with migration and checkpointing(Ipcpm).
- Initial placement with migration and without checkpointing(Ipm).

Local execution

In this case the applications are executed on the machine they have arrived at without any load balancing. Execution is done on a single machine. As we have discussed in the previous chapters each application has an application agent. In this case the application agents communicate with the LBFTS as each workstation has the full system running. However no load balancing is actually taking place. Note that the applications are not parallelised for this experiment. Comparing Le with Se we can estimate the overhead due to LBFTS.

Initial placement without migration and checkpointing

In this case the applications are parallelised and are subjected to Job entry load balancing by which they may get executed on different machines. They are not checkpointed and once started they run to completion on the machine they have been assigned to, i.e., they are not migrated from one machine to another.

Comparison of this case with the sequential execution cases provides the speed up due to initial placement.

Initial placement with checkpointing and without migration

In this case the applications are checkpointed at different intervals {30,60,90,120,180 seconds}. For different intervals we have different results.

Though this case does not involve migration, the checkpointing is used to provide fault tolerance. Note that no faults are exerted in this class of experiments. The results give an estimate of the overhead due to checkpointing.

Initial placement with migration and checkpointing

The applications are subjected to Jlb and they may get executed on different machines. Different machines over the network have their loads according to the job assignment done by Jlb. If these machines experience an imbalance later and if they find lightly loaded machines, the applications are migrated.

Applications that are migrated get restored from their recent checkpoints. Comparing this case with sequential execution cases we estimate the speed up achieved because of operation of full LBFTS. Comparing with Ipcp we estimate the improvement due to migration over initial placement, if any.

Initial placement with migration and without checkpointing

The overhead due to checkpointing has an effect on the completion times of the applications. Checkpointing should not be done in this case, as fault tolerance is not an issue. In case of migration the applications are checkpointed first and then migrated.

5.2.3 Speedup

One of the aim of performance analysis of LBS is to measure the speedup. As mentioned earlier speedup is measured using two different formulas (S1 and S2), based

on the experimentation done.

S1: If we consider a group of processes, each process can be considered as an individual application which has its own completion time. The average completion time of the application(a group of processes) is found by adding the completion times of all processes and averaging over the number of processes. This method is applied to Ts_Sm as well as the load balanced cases. The average completion time of the applications in Ts_Sm execution is divided by the average completion time of the applications in load balanced case to compute the speedup.

$$S1 = \Sigma Ts\_Sm\_Comptime_i/n \ / \ \Sigma Loadbalanced\_comptime_i/n$$

where i is the number of processes in the parallel application.

S1 provides a practical speedup for a timesharing system like LINUX. Theoretically this does not provide a correct speedup as it includes the effect of timesharing and as it depends on the order, the processes enter the system.

S2: To estimate the theoretically correct speedup, sequential execution time of the application has to be computed. For an application which is in the form of a group of processes, all the 20 or 30 processes are combined into a single process (Se case). For the parallel(LB) case the application is partitioned into 20 or 30 different processes and the completion time of the application is computed as the difference

between the first process entry time to the last process termination time. Thus, speedup is calculated as given in the formula below.

S2 = Se_Comptime / (Lastproc_Terminationtime - Firstproc_Entrytime)

As LBFTS is meant for a network of workstations open to general public S1 becomes a reasonable measure for speedup. Many experiments done on hypothetical applications have the speedup estimation according to the first formula(S1). However, it is important to observe the theoretical speedup and experimentation is done to estimate the same.

5.2.4 Setup for FTS

FTS is mainly responsible for restoring failed applications. The completion times of applications in case of machine failures have to be measured. The cases used are listed below:

- Ts_Sm.
- Local execution with failures(Le).
- Ipcpm without failures.
- Ipcpm with failures.

Ts_Sm

This case is similar to the one discussed in the LBS setup.

Local execution with failures

The applications are parallelised and executed on a single machine. They are checkpointed frequently. A failure is created on that machine and the applications get restored on the responsible machine.

Comparing this case with sequential execution, we estimate the time taken to restore applications on the system without performing any load balancing. The overhead due to LBFTS adds to the overhead due to fault detection and application restoration.

Ipcpm without failures

This case is similar to the one discussed in the setup for LBS. The applications are run in a full LBFTS environment.

Ipcpm with failures

All applications experience the same environment as in the previous case. In addition, random failures are injected into the network. Machines comeup and go down in a random way. The completion times of applications in such case are measured and compared with Ipcpm without failures.

5.3 Results and Discussion

Experiments are conducted according to the setup discussed in the previous section. Thus the results from those experiments are analyzed.

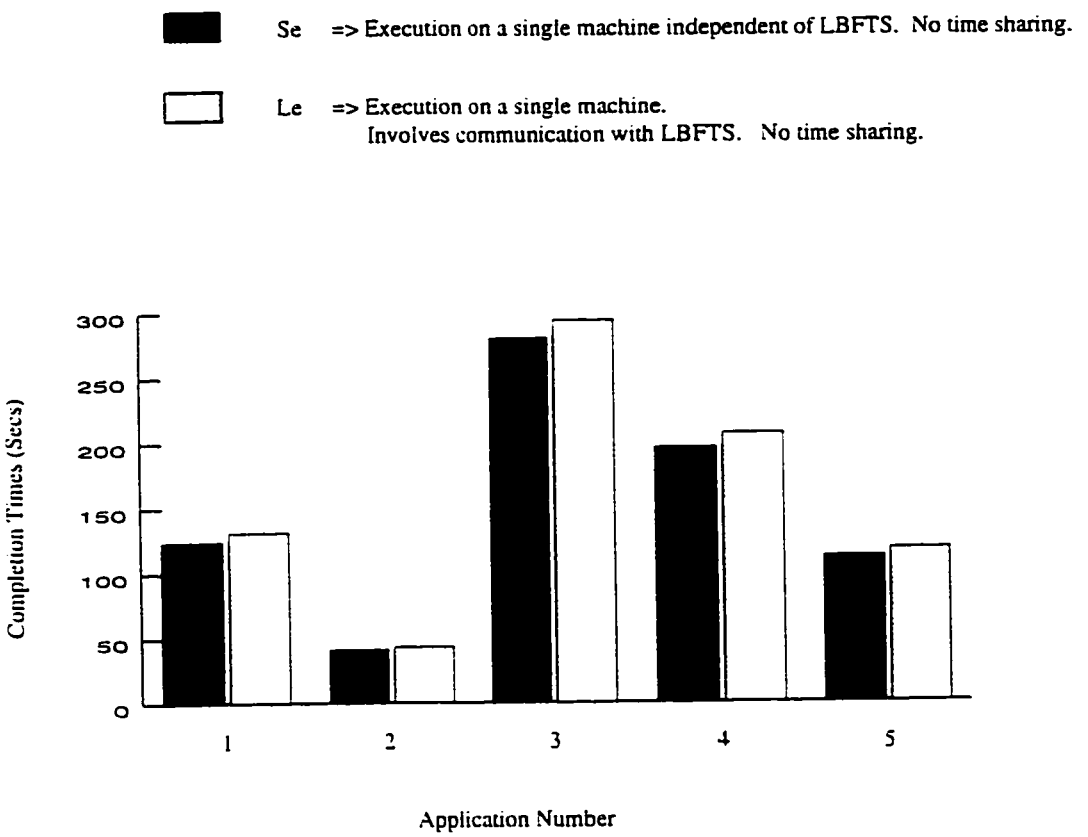
5.3.1 Overhead due to LBS

Se versus Le: The overhead involved because of LBFTS is observed by comparing the sequential and local executions. Table 5.1 shows the completion times of the five applications. In Figure 5.1 the completion times are plotted in the form of a bar chart. From the figure we can see five independent applications. Each

Application	Se	Le
1	124	131
2	41	43
3	280	294
4	196	207
5	112	118

Table 5.1: Cpu Intensive :: Se Vs. Le

application is submitted to the system separately i.e., it does not involve any time sharing overhead because of other user processes except the time sharing involved due to system processes. This provides the way to estimate the actual overhead due to LBFTS for each process. On an average the estimated overhead is 8 seconds per process which is a constant(1.05 % of the sequential time) as can be noticed from the figure. This overhead is accounted for communication between the application



Average Overhead Per process => 8 seconds

Figure 5.1: Cpu Intensive(No checkpoint) :: Se Vs. Le

agents and LBFTS. Also there is communication between the various daemons of LBFTS. Obviously, longer the application lesser the overhead in percentage of the sequential completion time.

5.3.2 Analysis of LBS:S1 (Average Completion Times)

CPU intensive applications

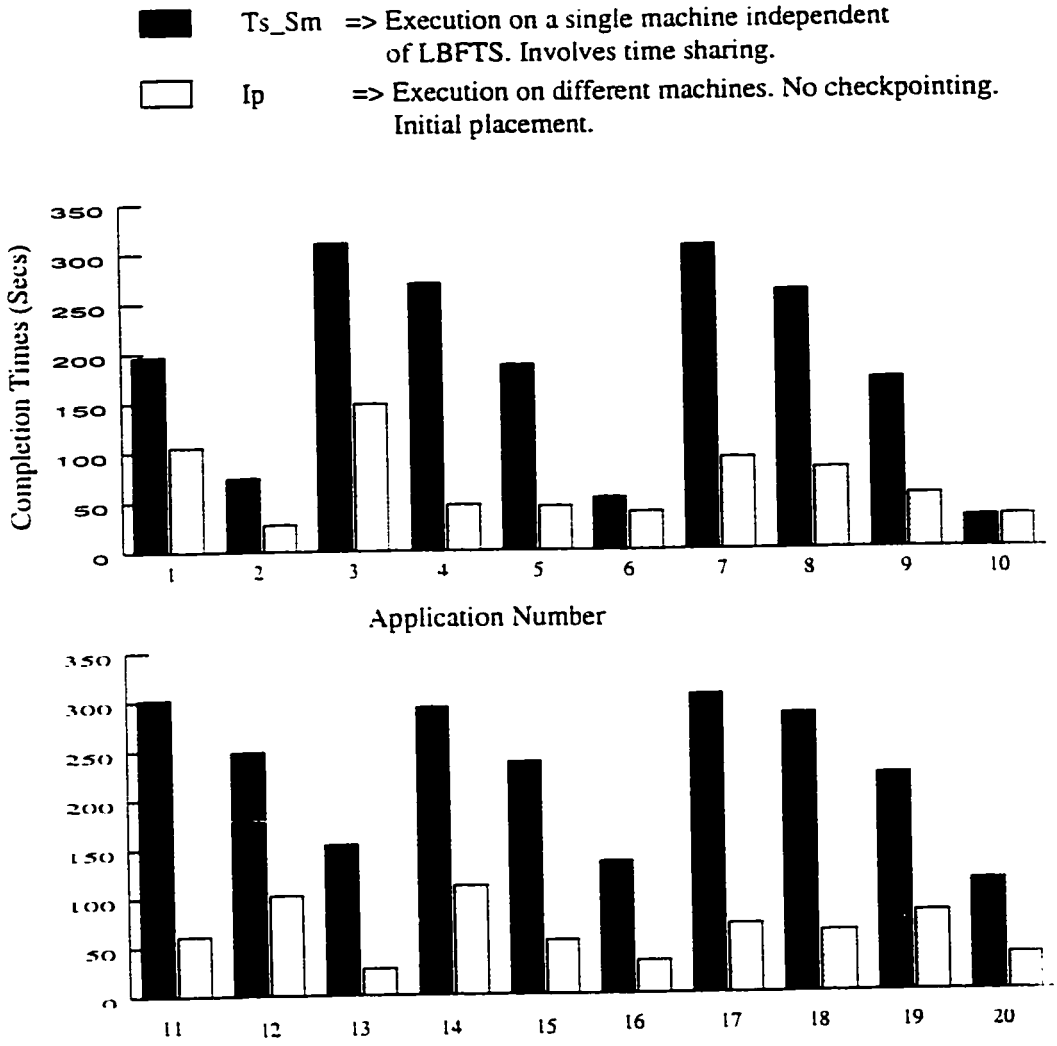
Ts_Sm versus Ip: The sequential case involves a group of processes running in a time sharing environment on a single machine. Speedup is measured using the formula S1. For an initial placement case the group of processes encounter the same time sharing environment, but on different machines. Hence the overhead due to time share factor in the case of initial placement is less, further reducing the completion times. Table 5.2 and Figure 5.2 depict the completion times and the speedup attained.

Initial placement without migration and checkpointing has a considerable improvement over the Ts_Sm case. The completion times are 3.17 times better than the sequential case. Here we need to mention that the number of machines that are connected over the network are three. For a network of 3 machines we are getting a speedup of 3.1 which can be explained, if we eliminate the overhead due to time sharing. For example in the case of n processes on a single machine, each process will take one slot in every n slots. Therefore if a process takes m slots of cpu time to

execute, its completion time will be m.n slots. In our case n is reduced to n/3, thus the completion time of a process gets reduced to m.n/3. From this we can notice the effect of time sharing in Ts_Sm case.

Application	Ts_Sm (Secs)	Ip (Secs)
1	197	105
2	74	27
3	310	148
4	269	46
5	187	44
6	53	38
7	306	92
8	260	81
9	171	54
10	31	32
11	302	61
12	249	102
13	154	27
14	293	111
15	237	55
16	135	33
17	304	70
18	284	62
19	222	81
20	113	37

Table 5.2: Cpu Intensive (No checkpoint):: Ts_Sm Vs. Ip



For a group of 20 independent CPU intensive processes =>

Speedup(S1)

Ts_Sm Versus Ip :: 3.17

Figure 5.2: Cpu Intensive(No checkpoint) :: Ts_Sm Vs. Ip

Ts_Sm versus Ipcp versus Ipcpm: Ipcp and Ipcpm have a similar working environment as of Ip. Table 5.3 presents the completion times for a group of 20 processes of an application. Figure 5.3 shows the comparison between Ts_Sm execution and the load balancing cases Ipcp and Ipcpm. In Ipcp and Ipcpm the applications are checkpointed frequently i.e, every 60 seconds. Sequential execution does not involve any checkpointing overhead.

Eventhough the checkpointing overhead is involved we can notice that ... Ipcp and Ipcpm are 2.68 and 2.63 times better than Ts_Sm, respectively. In this we can notice the improvement due to Plb over Jlb. For a time consuming process we have a considerable improvement.

Table 5.4 provides the completion times for 90 seconds checkpointing case. Figures 5.4,5.5 are similar to Figure 5.3, except that the checkpointing periods are 90 and 120 seconds respectively. From this we can notice a speedup of 2.91 in both Ipcp and Ipcpm.

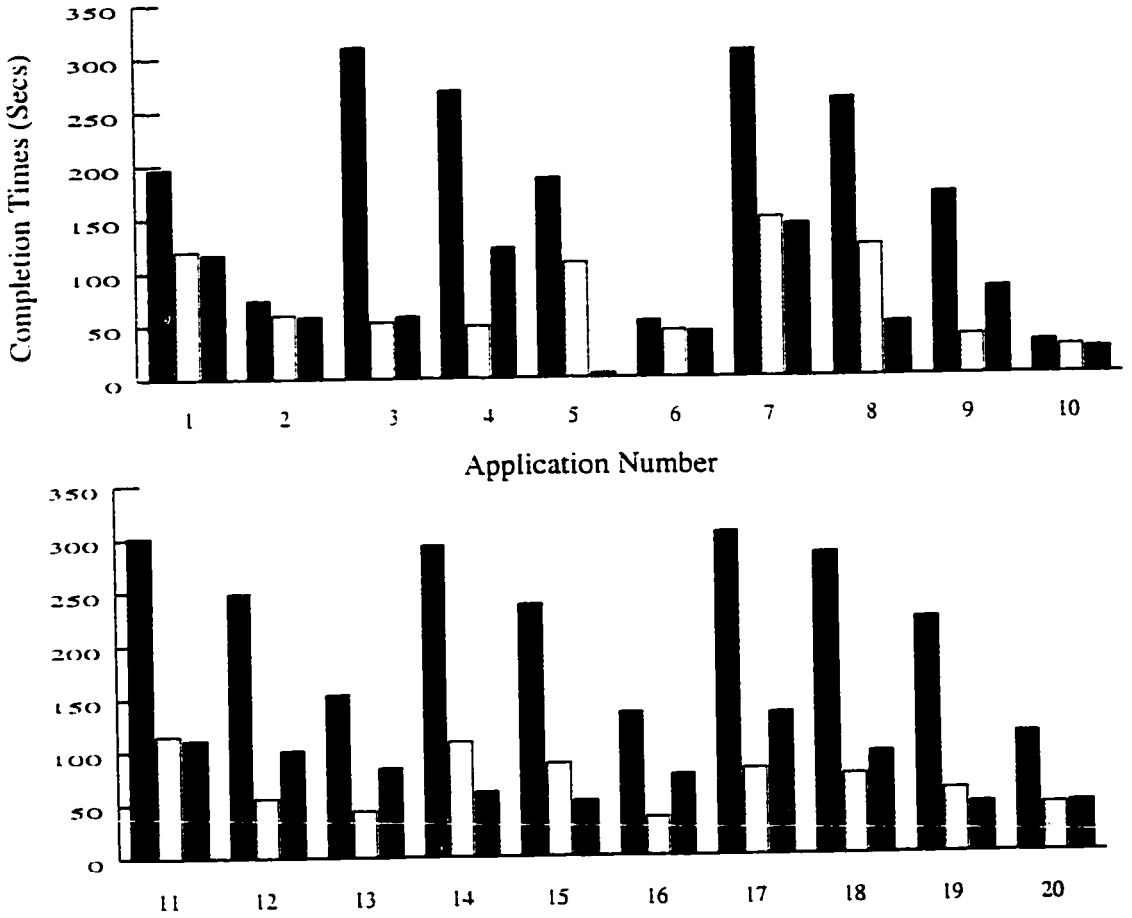
Application	Ts_Sm	Ipcp	Ipcpm
1	197	120	117
2	74	60	58
3	310	53	58
4	269	49	122
5	187	108	4
6	53	44	43
7	306	149	143
8	260	123	50
9	171	37	82
10	31	26	24
11	302	115	111
12	249	56	101
13	154	44	84
14	293	108	61
15	237	87	52
16	135	36	76
17	304	81	134
18	284	75	96
19	222	60	47
20	197	120	117

Table 5.3: Cpu Intensive(Checkpoint 60):: Ts_Sm Vs. Ipcp Vs. Ipcpm

Ts_Sm => Execution on a single machine independent of LBFTS. Involves time sharing.

Ipcp => Execution on different machines. Periodic check. Initial placement.

Ipcpm => Execution on different machines. Periodic check. Initial placement & migration.



For a group of 20 independent CPU intensive processes =>

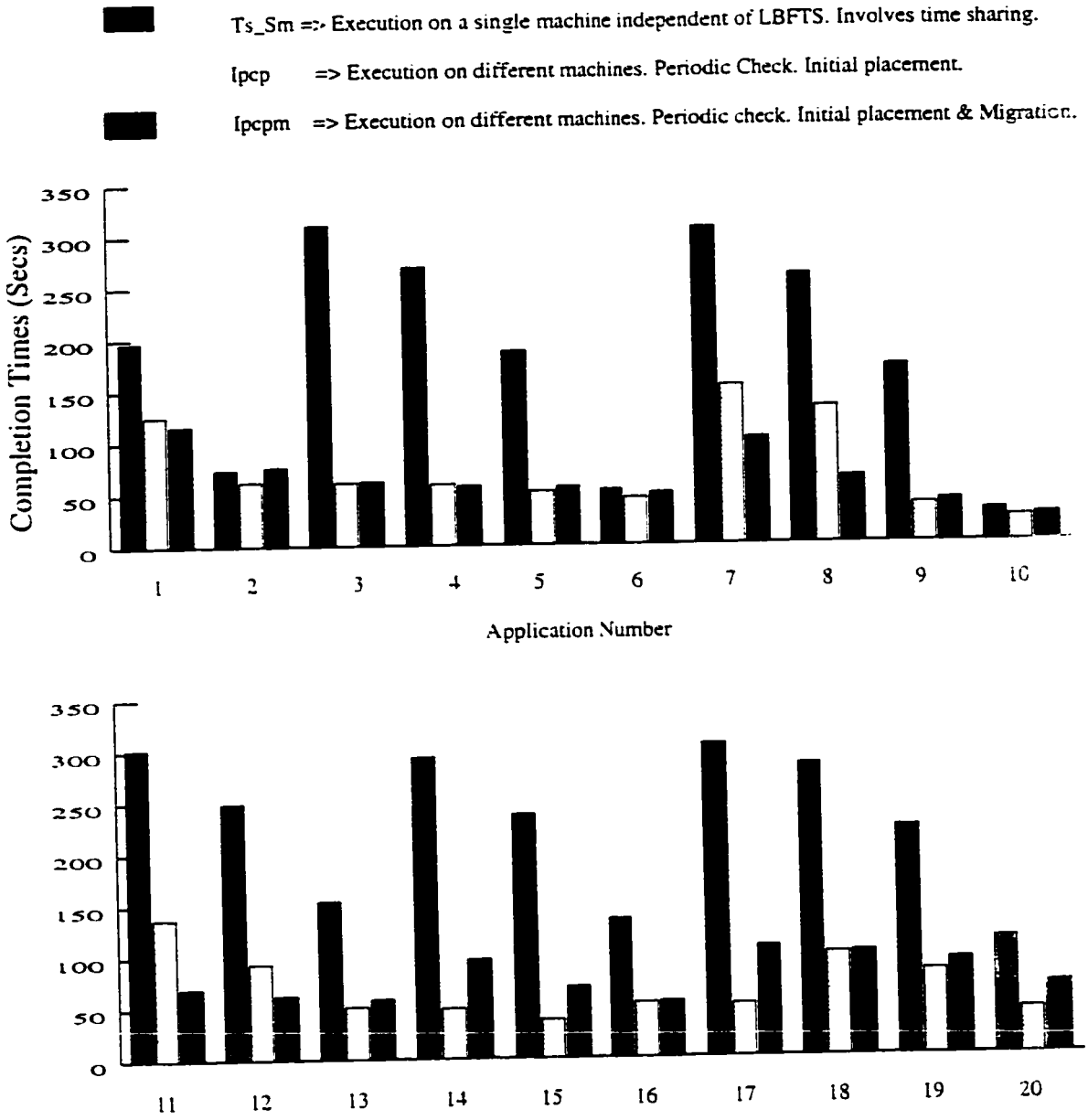
Speedup(S1)

Ts_Sm vs Ipcp :: 2.68
Ts_Sm vs Ipcpm :: 2.63

Figure 5.3: Cpu Intensive :: Ts_Sm Vs Ipcp Vs Ipcpm: Checkpoint 60 secs

Application	Ts_Sm	Ipcp	Ipcpm
1	197	125	116
2	74	62	76
3	310	61	62
4	269	59	57
5	187	52	56
6	53	45	50
7	306	153	102
8	260	132	64
9	171	37	41
10	31	24	26
11	302	137	69
12	249	93	62
13	154	51	58
14	293	49	96
15	237	37	69
16	135	53	54
17	304	51	107
18	284	100	101
19	222	82	93
20	113	44	70

Table 5.4: Cpu Intensive(Checkpoint 90):: Ts_Sm Vs. Ipcp Vs. Ipcpm



For a group of 20 independent CPU intensive processes =>

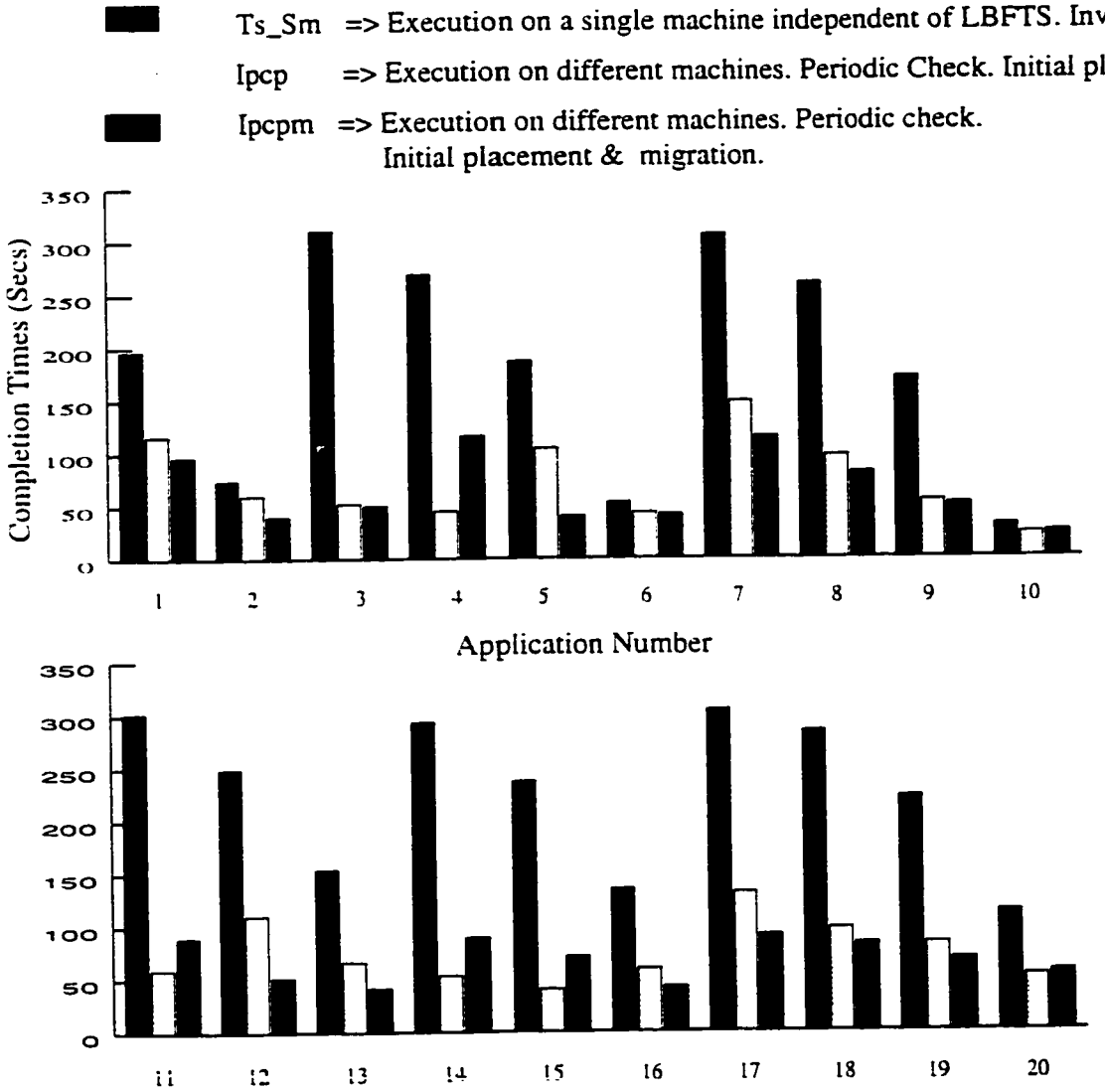
Speedup(S1)

Ts_Sm vs Ipcp :: 2.91
Ts_Sm vs Ipcpm :: 2.91

Figure 5.4: Cpu Intensive :: Ts_Sm Vs. Ipcp Vs. Ipcpm: Checkpoint 90 secs

Application	Ts_Sm	Ipcp	Ipcpm
1	197	116	96
2	74	60	40
3	310	52	50
4	269	45	116
5	187	104	40
6	53	43	41
7	306	148	115
8	260	97	81
9	171	54	51
10	31	23	24
11	302	59	89
12	249	110	51
13	154	66	41
14	293	53	89
15	237	40	71
16	135	59	42
17	304	131	91
18	284	97	83
19	222	83	68
20	113	52	56

Table 5.5: Cpu Intensive(Checkpoint 120):: Ts_Sm Vs. Ipcp Vs. Ipcpm



For a group of 20 time sharing CPU intensive processes =>

Speedup(S1)

Ts_Sm vs Ipcp :: 2.8
Ts_Sm vs Ipcpm :: 3.1

Figure 5.5: Cpu Intensive :: Ts_Sm Vs. Ipcp Vs. Ipcpm: Checkpoint 120 secs

Ts_Sm versus Ip versus Ipm: A group of 5 independent cpu intensive and long running applications are used to see the effect of LBS. Table 5.6 provides the completion times. Figure 5.6 depicts the comparisons between Ts_Sm .Ip and Ipm. S1 is used to measure the speedup.

Application	Ts_Sm	Ip	Ipm
1	4964	2272	1858
2	2914	568	381
3	7111	4051	4207
4	6320	4845	1718
5	4619	2166	2042

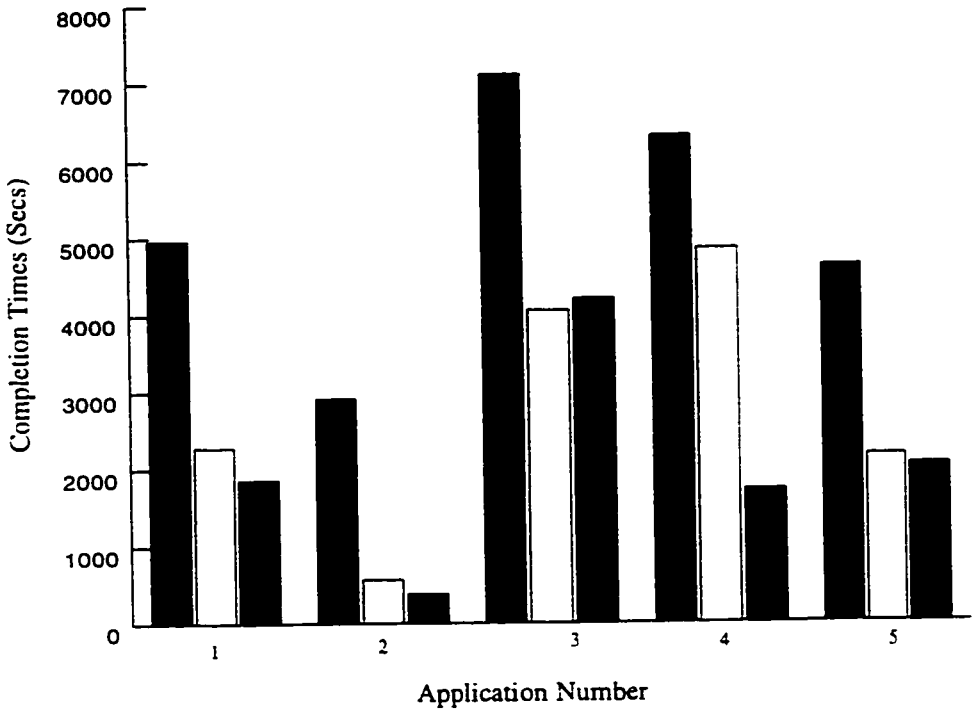
Table 5.6: Cpu Intensive :: Ts_Sm Vs. Ip Vs. Ipm

Initial placement is 1.9 times better than the sequential case(Speedup of 1.9) and migration is 2.54 times better. From this figure we can notice that migration case performance is 1.34(2.54/1.9) times better than Ip. The previous figures demonstrated small applications having completion times in the range 20-300 seconds. In those applications there was no significant difference between migration and Initial placement cases. In this experiment we have processes that have completion times in the range 3000-7500. From these results we can conclude that long running applications are more suitable for migration and Initial placement is enough for small applications.

Ts_Sm => Execution on a single machine independent of LBFTS. Involves time sharing.

Ip => Execution on different machines. No check. Initial placement.

Ipm => Execution on different machines. No check. Initial placement & migration.



For a group of 5 independent CPU intensive processes =>

Speedup(S1)

Ts_Sm vs Ip :: 1.9
Ts_Sm vs Ipm :: 2.54

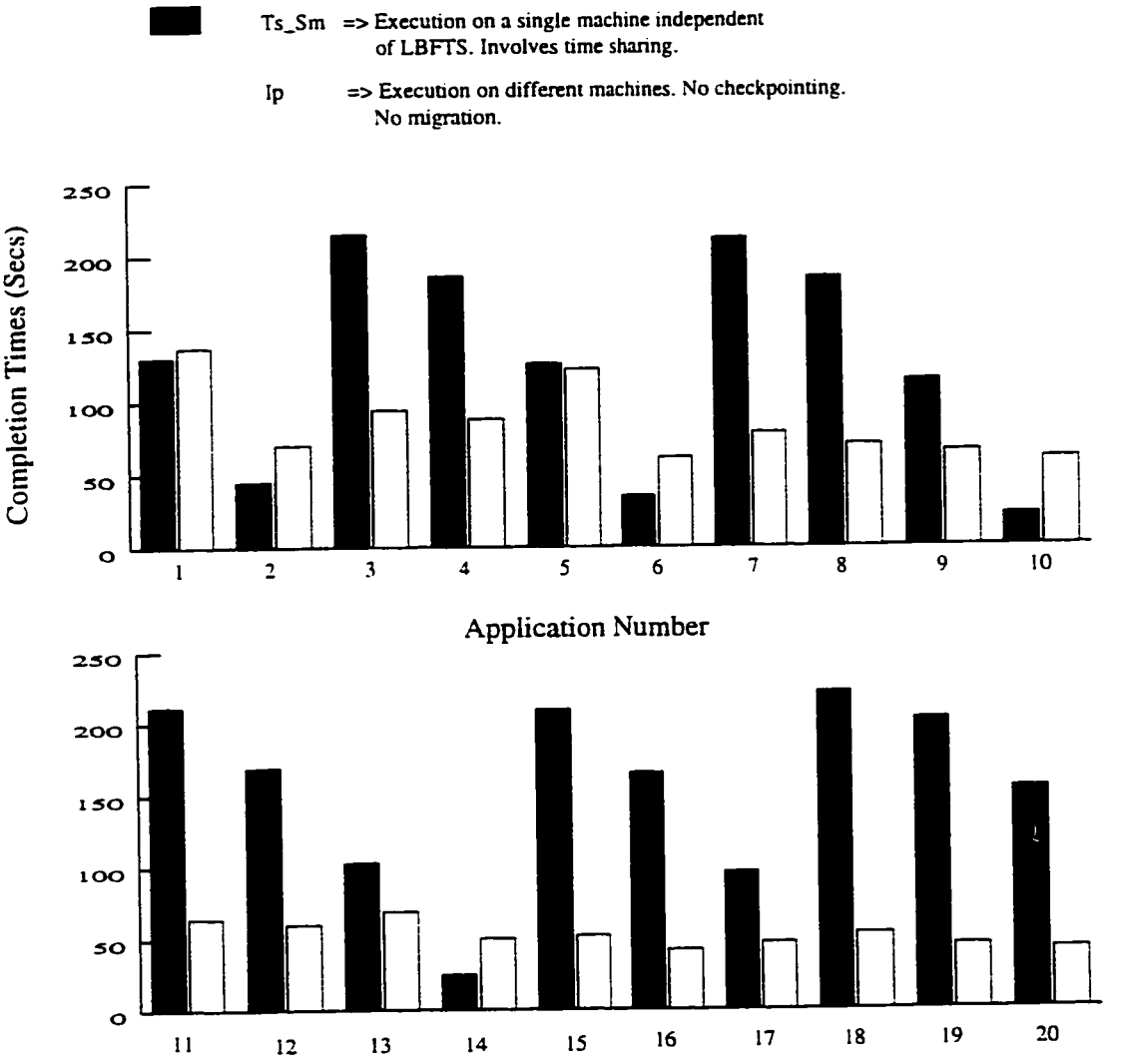
Figure 5.6: Cpu Intensive(No checkpoint) :: Ts_Sm Vs. Ip Vs. Ipm

IO intensive applications

Ts_Sm versus Ip: Table 5.7 provides a sample of 20 values from a group of 30 independent applications. Comparison of sequential execution with initial placement without checkpointing and migration is depicted in Figure 5.7. Sequential and Initial placement have the same time sharing environment as of the cpu intensive applications. The average of I/O(read and write) for these processes was 10000. Each IO involves 250 bytes of data.

Application	Ts_Sm(Secs)	Ip(Secs)
1	130	137
2	45	70
3	215	94
4	186	88
5	126	122
6	35	61
7	212	78
8	185	70
9	114	65
10	22	60
11	211	64
12	169	60
13	103	69
14	25	50
15	209	52
16	165	42
17	96	47
18	221	53
19	202	45
20	154	42

Table 5.7: Io intensive :: Ts_Sm Vs. Ip



For a group of 30 independent IO intensive processes =>

Speedup(S1)

Ts_Sm vs Ip :: 2.5

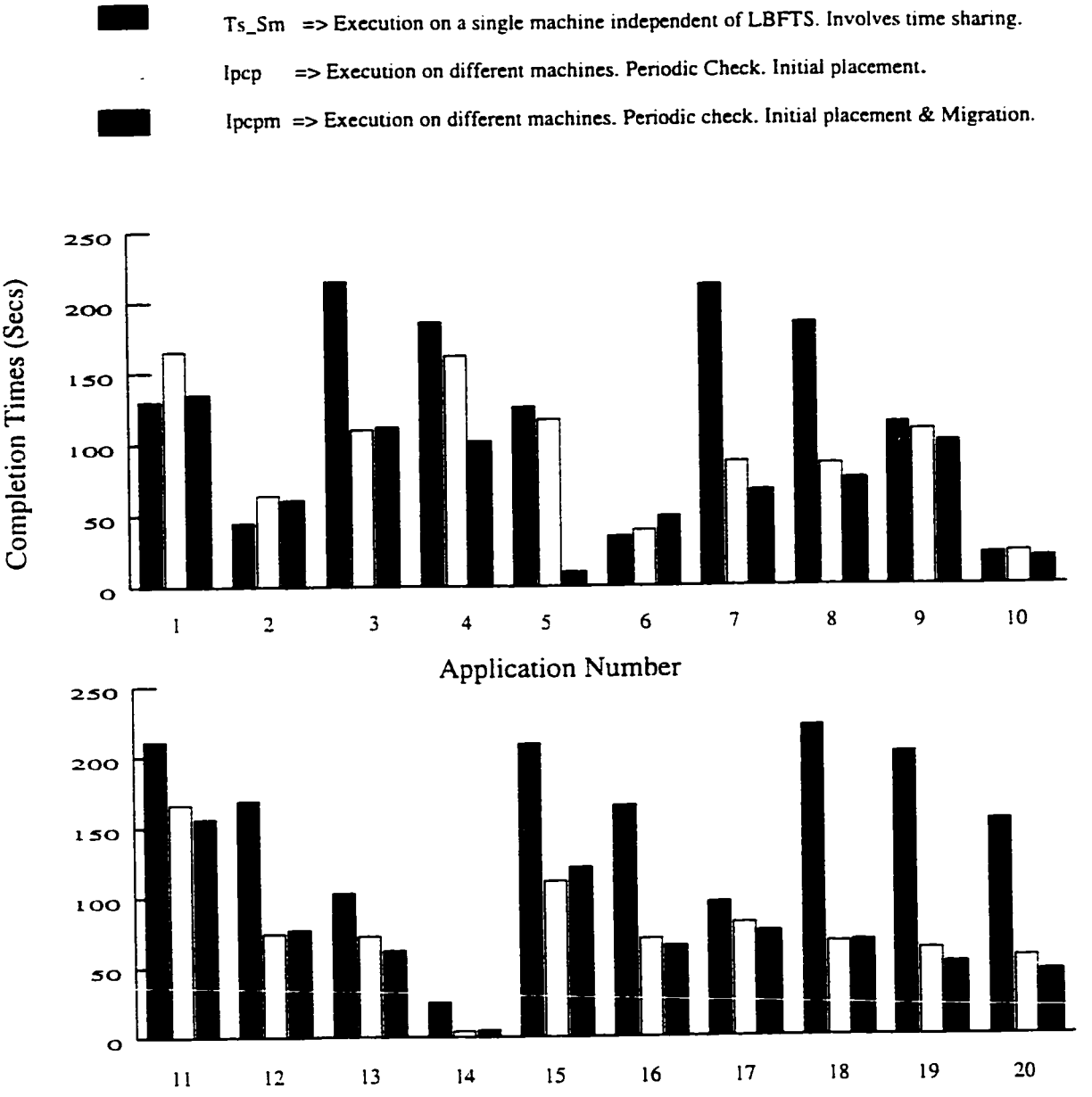
Figure 5.7: Io Intensive(No checkpoint) :: Ts_Sm Vs. Ip

The results show that initial placement is 2.5 times better than the sequential case. The speedup has been measured using S1, i.e., ratio of average completion times.

Ts_Sm versus Ipcp versus Ipcpm: In this case the applications are checkpointed frequently i.e, every 60 seconds. Sequential execution does not involve any checkpointing overhead. We can notice that Ipcp and Ipcpm are 1.7 and 1.9 times better than the sequential case, respectively. The results demonstrates an improvement for the migration of applications when compared with the initial placement, in some cases. A sample of completion times is given in Table 5.8. The results are plotted in Figure 5.8.

Application	Ts_Sm	Ipcp	Ipcpm
1	130	165	135
2	45	64	61
3	215	110	112
4	186	162	102
5	126	117	10
6	35	39	49
7	212	87	67
8	185	85	75
9	114	109	101
10	22	23	19
11	211	166	156
12	169	74	77
13	103	72	62
14	25	4	5
15	209	111	121
16	165	70	65
17	96	81	75
18	221	67	68
19	202	62	52
20	154	56	46

Table 5.8: Io intensive(Checkpoint 60):: Ts_Sm Vs. Ipcp Vs. Ipcpm



For a group of 30 independent IO intensive processes =>

Speedup(S1)

Ts_Sm vs Ipcp :: 1.7
Ts_Sm vs Ipcpm :: 1.9

Figure 5.8: Io Intensive :: Ts_Sm Vs. Ipcp Vs. Ipcpm: Checkpoint 60 secs

Communicating applications

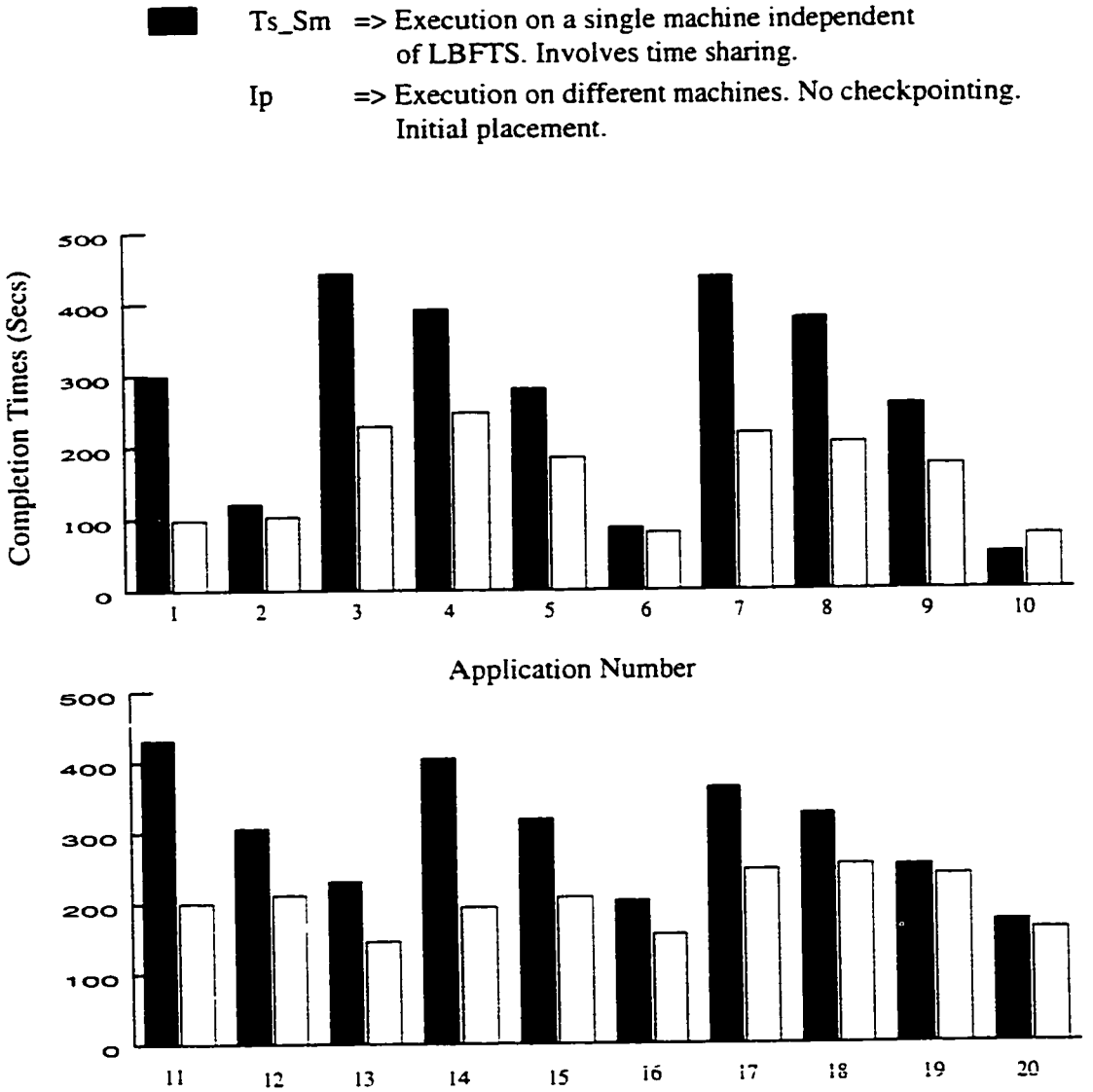
Ts_Sm Versus Ip: The working environment is similar to the environment of the cpu intensive applications. Table 5.9 provides the results. The improvement gained over sequential execution by initial placement without migration and checkpointing can be depicted as in Figure 5.9. The completion times on an average are 1.6 times better than the sequential case. Here we have to remember that the communication intensive application involves different clients communicating to the same server. As there is a lot of centralized(server) execution the parallel application converges into a sequential application, hence the lowering of the speedup.

Ts_Sm Versus Ipcp Versus Ipcpm: Sequential execution is compared with the two load balancing cases. In Ipcp and Ipcpm the applications have checkpointing overhead as the tasks are checkpointed every 90 seconds. Eventhough the checkpointing overhead is involved we have observed that Ipcp and Ipcpm achieve a speedup of 1.32 and 1.39, respectively. There is an improvement due to migration, for large tasks, over the initial placement. Table 5.10 provides the completion times. Figure 5.10 plots the completion times for a group of 20 tasks of a parallel application.

Table 5.11 refers to the completion times used in the Figure 5.11, which is similar to Figure 5.10 except that the checkpointing period is 120 seconds. In this

Application	Ts_Sm(Secs)	Ip(Secs)
1	299	98
2	121	103
3	442	228
4	392	247
5	281	185
6	87	80
7	437	218
8	380	205
9	258	174
10	50	75
11	431	200
12	306	211
13	231	146
14	405	194
15	319	208
16	203	155
17	363	246
18	326	253
19	252	238
20	173	161

Table 5.9: Communication intensive :: Ts_Sm Vs.Ip.



Parallel application: 20 Tasks =>

Speedup (S1)

Ts_Sm vs Ip :: 1.6

Figure 5.9: Communication Intensive(No checkpoint) :: Ts_Sm Vs. Ip

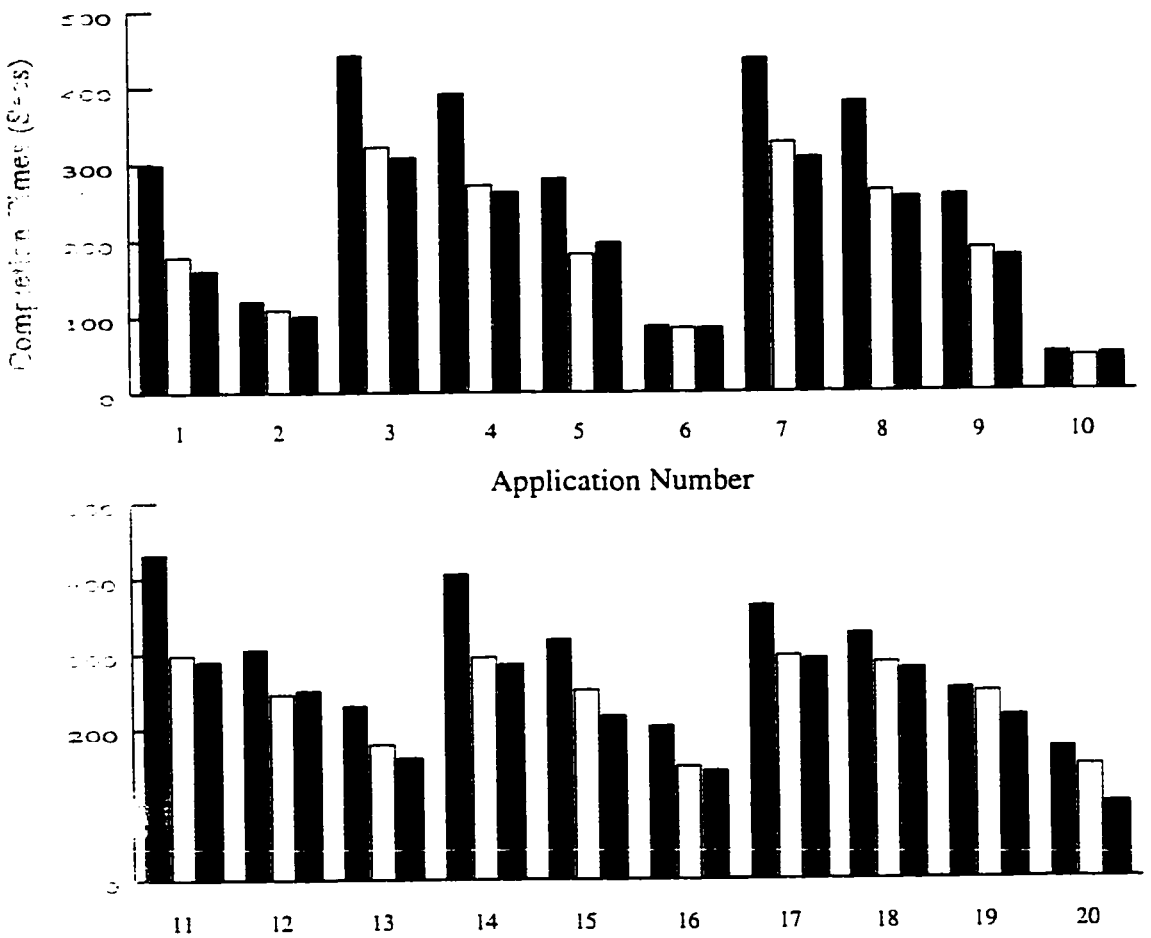
Application	Ts_Sm	Ipcp	Ipcpm
1	299	179	161
2	121	109	101
3	442	322	308
4	392	272	263
5	281	182	197
6	87	84	84
7	437	327	307
8	380	264	255
9	258	188	178
10	50	45	48
11	431	298	290
12	306	246	251
13	231	179	161
14	405	295	286
15	319	251	217
16	203	149	143
17	363	296	292
18	326	287	279
19	252	247	215
20	173	149	99

Table 5.10: Communication intensive(Checkpoint 90):: Ts_Sm Vs. Ipcp Vs. Ipcpm

Ts_Sm => Execution on a single machine independent of LBFTS. Involves time sharing.

Ipcp => Execution on different machines. Periodic Check. Initial placement.

Ipcpm => Execution on different machines. Periodic check.
Initial placement & Migration.



Parallel application : 20 tasks =>

Speedup(S1)

Ts_Sm vs Ipcp :: 1.32
Ts_Sm vs Ipcpm :: 1.39

Figure 5.10: Communication Intensive :: Ts_Sm Vs. Ipcp Vs. Ipcpm: Checkpoint 90 secs

case the speedups for Ipcp and Ipcpm are 1.44 and 1.58. respectively.

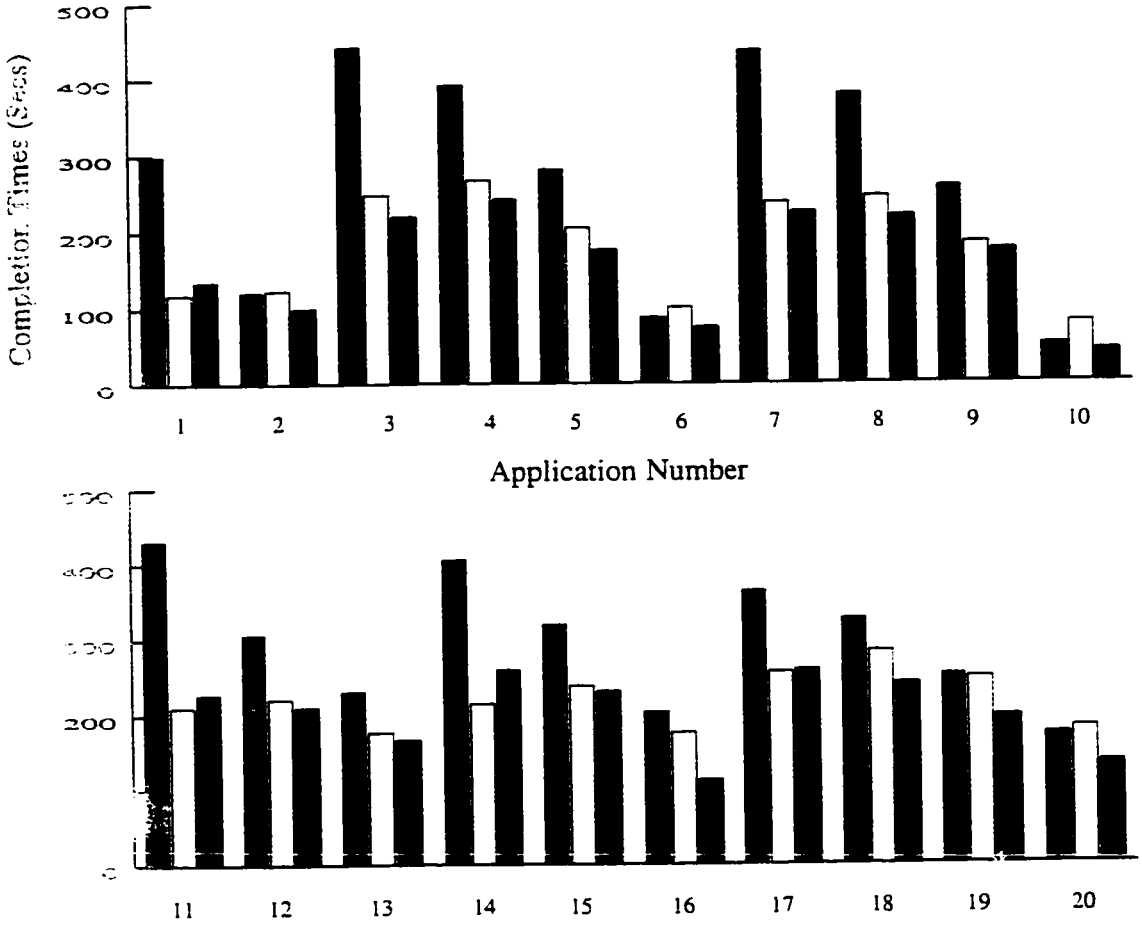
Application	Ts_Sm	Ipcp	Ipcpm
1	299	118	134
2	121	123	99
3	442	248	220
4	392	267	242
5	281	205	176
6	87	100	74
7	437	238	225
8	380	245	220
9	258	184	175
10	50	79	42
11	431	210	227
12	306	221	210
13	231	176	166
14	405	214	259
15	319	238	231
16	203	175	112
17	363	256	259
18	326	283	241
19	252	248	197
20	173	181	135

Table 5.11: Communication(Checkpoint 120):: Ts_Sm Vs. Ipcp Vs. Ipcpm

Ts_Sm => Execution on a single machine independent of LBFTS. Involves time sharing.

Ipcp => Execution on different machines. Periodic Check. Initial placement.

Ipcpm => Execution on different machines. Periodic check.
Initial placement & Migration.



Parallel applicaitons:: 20 Tasks =>

Speedup(S1)

Ts_Sm vs Ipcp :: 1.44
Ts_Sm vs Ipcpm :: 1.58

Figure 5.11: Communication Intensive :: Ts_Sm Vs. Ipcp Vs. Ipcpm: Checkpoint 120 secs

5.3.3 Analysis of LBS: S2 (Completion times)

Independent CPU intensive

Table 5.12 provides the results and Figure 5.12 plots the results for cpu intensive applications. From the figure we can see three different cases. The first is a sequential case where the parallel application is executed as a single process, thus involving no time sharing overhead. The second case is Ip where the application is parallelised and the processes are run on different machines. The third case is Ipm which involves migrations. The completion time is calculated as the difference of the last process termination time to the first process entry time, as in the speedup formula S2. A speedup of 1.51 is achieved in this case. Migration is plotted in the third bar of the figure. The estimated speedup in the migration case is 1.44 which is less than the initial placement case, as we have said previously that short processes are not suitable for migration.

Run	Se	Ip	Ipm
1	272	180	189

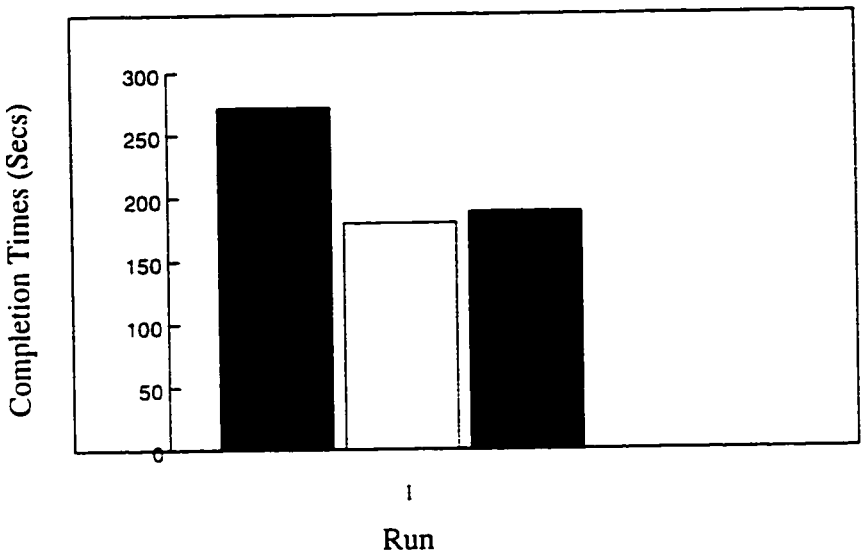
Table 5.12: Cpu intensive :: Se Vs.Ip Vs.Ipm

The speedup S2 can be compared with the S1 case. Using S1 a speedup of 3.17 (Figure 5.2) was achieved for the Ip setup. In our case the speedup is 1.51. From this we can see the effect of timesharing.

Se => Execution on a dedicated machine . No time sharing.

Ip => Execution on different machines. No check. Initial placement.

Ipm => Execution on different machines. No check.
Initial placement & migration.



For a group of 20 independent CPU intensive processes =>

Speedup(S2)

Se vs Ip :: 1.51
Se vs Ipm :: 1.44

Figure 5.12: Cpu Intensive(No checkpoint) :: Se(No time sharing) Vs. Ip Vs. Ipm

Matrix Multiplication

Matrix multiplication provides a real environment. In the experiment we have used different matrix sizes to analyze the various characteristics of the LBS. The sizes used are 100 x 100, 250 x 250, 400 x 400, 500 x 500 and 600 x 600.

The completion time of the server which gathers the matrix multiplication results is used to analyze the system. The sequential case does not involve any time sharing overhead as the matrix multiplication is done by a single process. The speedup is calculated using the formula S2. Hence the speedup achieved in these cases is theoretically correct.

100 x 100 Matrix

Table 5.13 provides the completion times used in Figure 5.13 which plots the results for Se, initial placement without checkpointing and migration(Ip), initial placement with checkpointing and without migration(Ipcp). In the checkpointing case the tasks of the parallel application are checkpointed every 60 seconds. From the figure we can observe that the tasks do not experience the checkpointing overhead as they get completed within 60 seconds.

Comparison of sequential completion times with the two load balancing cases shows that sequential execution is better than the load balanced execution on a network of 3 workstations. We can conclude that load balancer has a negative effect on the execution of small matrices. This conclusion is expected, for cases where load

balancer overhead exceeds the benefit of concurrency.

Run	Se	Ip	Ipcp
1	9	45	46
2	8	43	37
3	8	39	41

Table 5.13: 100 x 100 matrix:: Se Vs. Ip Vs. Ipcp

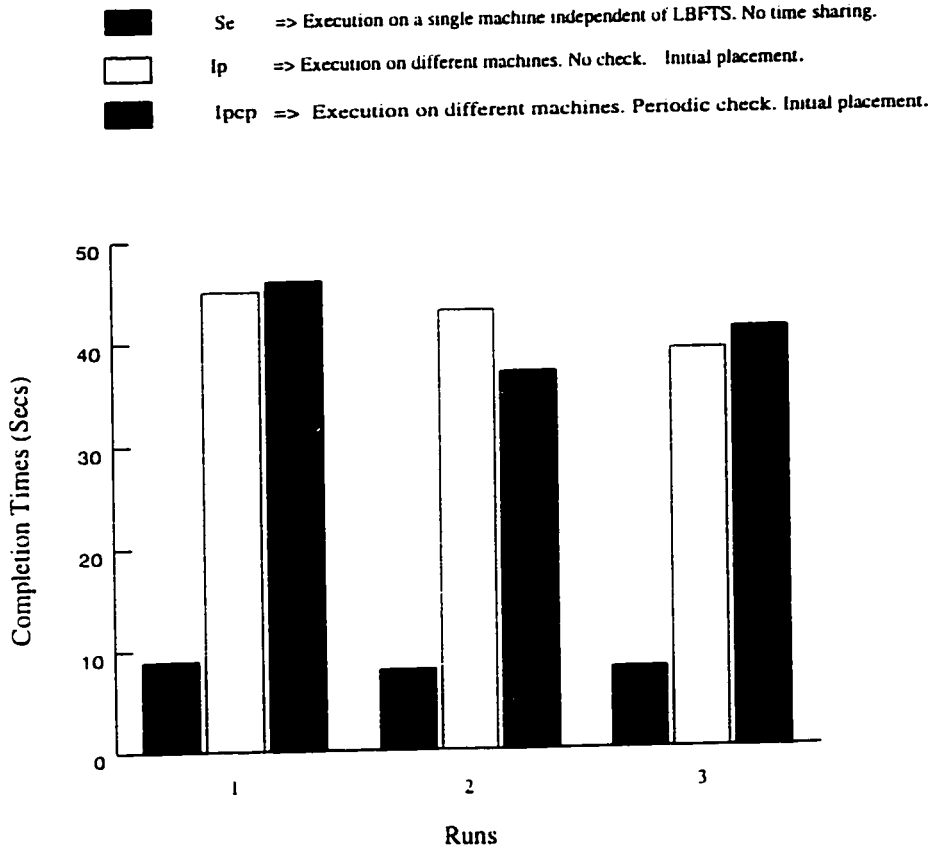
250 x 250 Matrix

In this case the applications experience the same environment as in the above case. Table 5.14 provides the completion times. Figure 5.14 depicts the results in the form of a histogram. In the figure we can see four completion times for each run i.e., for Se, Ip, Ipcp and Ipcpm. A speedup of 1.91 is achieved for Ip. In Ipcp and Ipcpm the speedups achieved are 1.96 and 2.08, respectively. The applications do not experience the checkpointing overhead as they get completed within 60 seconds, hence the same speedup in all non checkpointing and checkpointing cases. The

Run	Se	Ip	Ipcp	Ipcpm
1	93	50	47	41
2	90	46	46	47

Table 5.14: 250 x 250 matrix:: Se Vs. Ip Vs. Ipcp Vs. Ipcpm

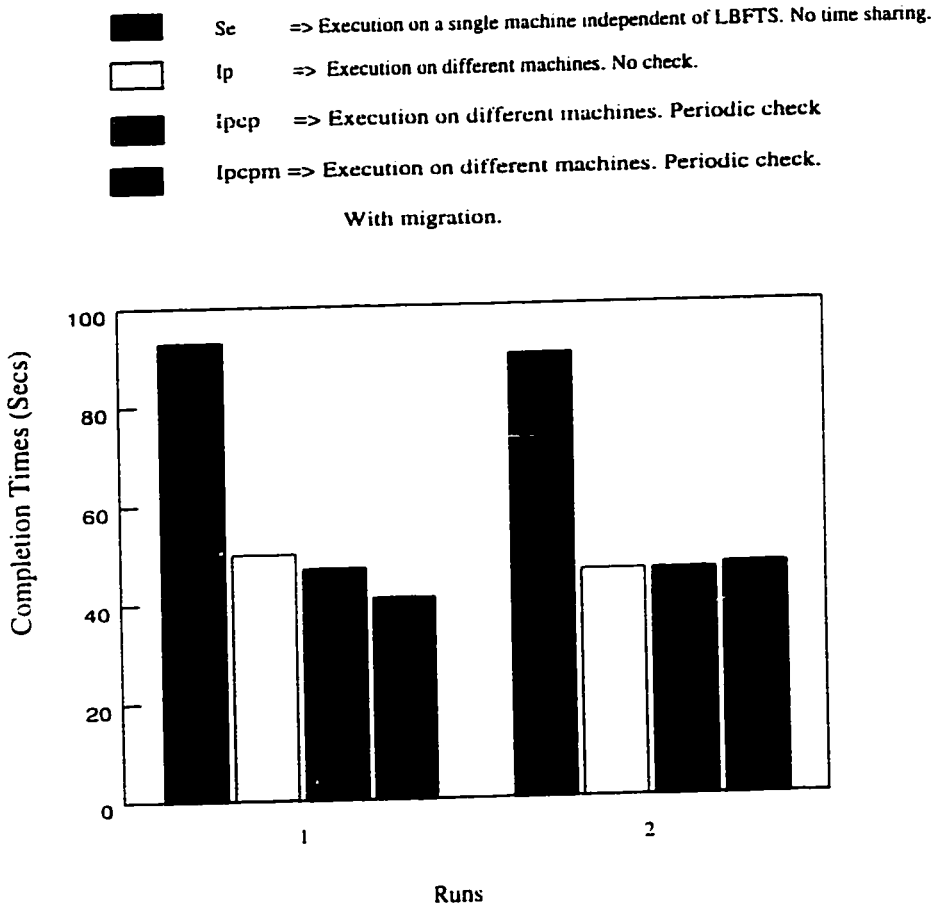
load balancer has a positive effect on the multiplication of a 250 x 250 matrix. This



Speedup(S2)

In this case speedup < 1.

Figure 5.13: 100 x 100 Matrix: Checkpoint 60 secs



Speedup (S2)

Se vs Ip	::	1.91
Se vs Ipcp	::	1.96
Se vs Ipcpm	::	2.08

Figure 5.14: 250 x 250 Matrix: Checkpoint 60 secs

can be seen by comparing these results with those of a 100 x 100 matrix.

400 x 400 Matrix

Table 5.15 shows the results. In multiplication of a 400 x 400 matrix the LB shows further improvement in the completion times as depicted in Figure 5.15. Ip achieves a speedup of 2.03. Ipcp achieves a speedup of 1.83 which is less than the Ip speedup as there is a checkpointing overhead involved in this case. Migration shows an improvement as it performs 2.14 times better than Se.

Run	Se	Ip	Ipcp	Ipcpm
1	380	190	203	175
2	382	185	215	181

Table 5.15: 400 x 400 matrix:: Se Vs. Ip Vs. Ipcp Vs. Ipcpm

500 x 500 Matrix

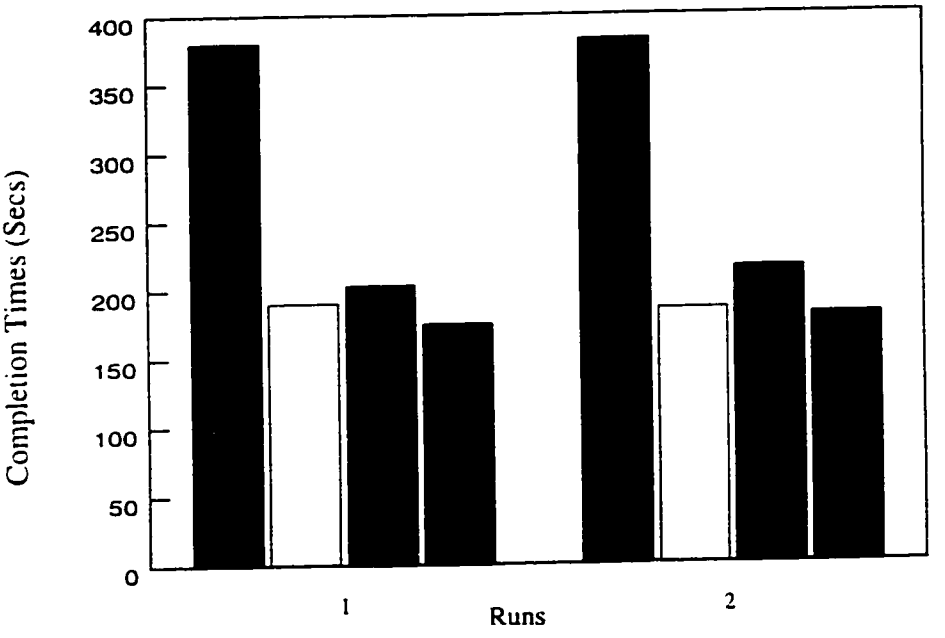
Table 5.16 provides the completion times for this case. In Figure 5.16 the results are plotted in the form of a bar chart. Ip and Ipcp achieve a speedup of 2.28 and 2.14. Migration case achieves a speedup of 2.27 which is a better than Ipcp case. The speedup in the Ip and Ipcpm setup for the 400 x 400 matrix were 1.63 and 1.81, respectively. So, as the matrix size increases the speed up increases, as we can see the improvement in the larger matrices.

Se => Execution on a single machine independent of LBFTS. No time sharing.

Ip => Execution on different machines. No check.

Ipcp => Execution on different machines. Periodic check. Initial placement.

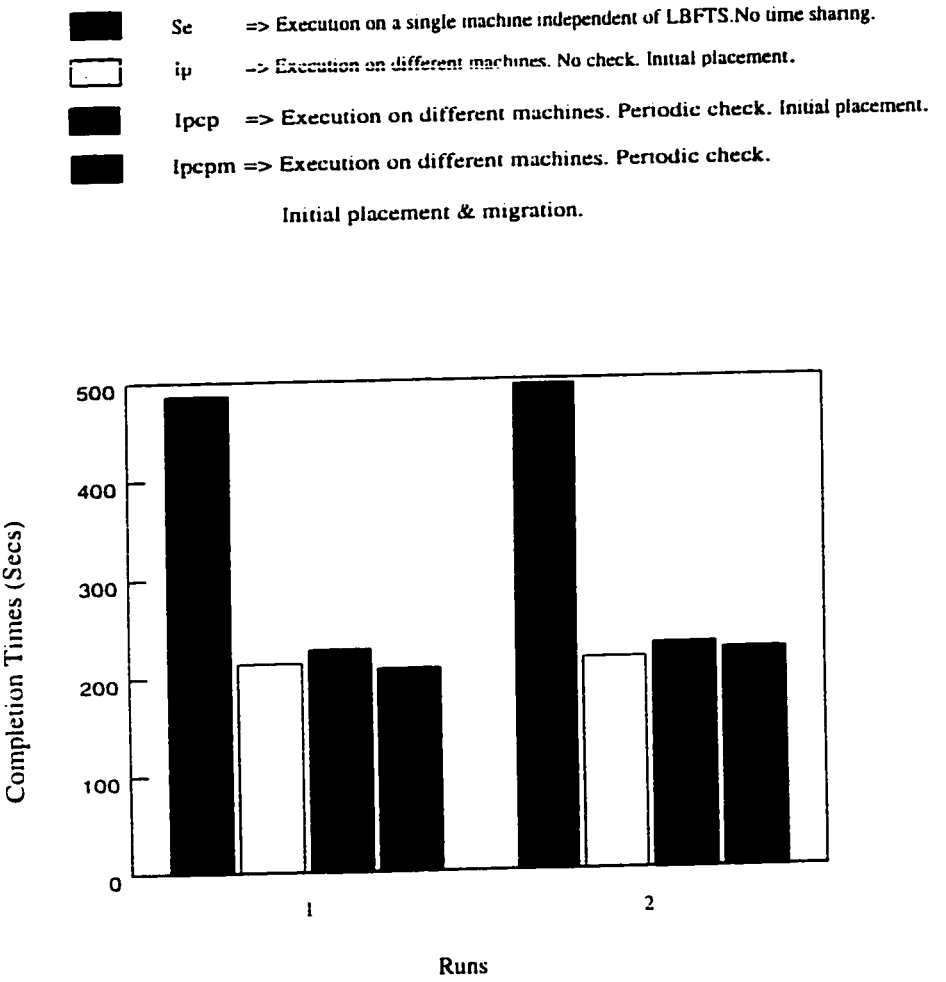
Ipcpm => Execution on different machines. Periodic check. Initial placement..



Speedup(S2)

Se vs Ip :: 2.03
Se vs Ipcp :: 1.83
Se vs Ipcpm :: 2.14

Figure 5.15: 400 x 400 Matrix: Checkpoint 60 secs



Speedup (S2)

Se vs Ip	::	2.28
Se vs Ipcp	::	2.14
Se vs Ipcpm	::	2.27

Figure 5.16: 500 x 500 Matrix: Checkpoint 60 secs

Run	Se	Ip	Ipcp	Ipcpm
1	487	214	228	208
2	495	216	230	224

Table 5.16: 500 x 500 matrix:: Se Vs. Ip Vs. Ipcp Vs. Ipcpm

600 x 600 Matrix

Table 5.17 provides the completion times. Figure 5.17 depicts the results in the form of a bar chart. Ip is 2.89 times better than the sequential case. Ipcp achieves a speedup of 2.69, whereas Ipcpm is 2.97 times better than the Se. From these results we can conclude that LBFTS shows better performance for larger applications, as we can see the speedup for the 500 x 500 matrix (2.28, 2.12, 2.27 for Ip, Ipcp and Ipcpm, respectively). Note that larger the matrices higher the cpu intensity. Hence the higher speedup. Eventhough the computation increases with the size of the matrices the communication does not change.

Run	Se	Ip	Ipcp	Ipcpm
1	1730	633	668	589
2	1842	625	662	615

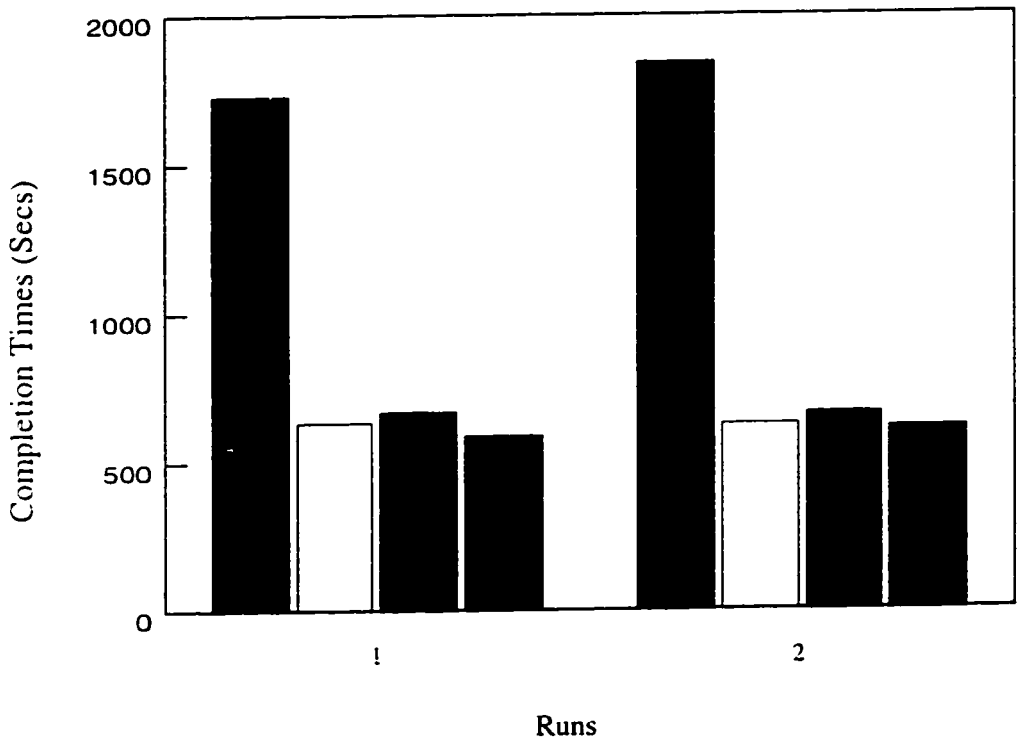
Table 5.17: 600 x 600 matrix:: Se Vs. Ip Vs. Ipcp Vs. Ipcpm

Se => Execution on a single machine independent of LBFTS. No time sharing

Ip => Execution on different machines. No check. Intial placement.

Ipcp => Execution on different machines. Periodic check. Initial placement.

Ipcpm => Execution on different machines. Periodic check.
Initial placement. & migration.



Speedup(S2)

Sequential vs Ip_Nocheck :: 2.84
Sequential vs Ip_Check :: 2.69
Sequential vs Mig_Check :: 2.97

Figure 5.17: 600 x 600 Matrix: Checkpoint 60 secs

5.3.4 Analysis of FTS

The system is analyzed by measuring the times taken for failed applications.

Ts_Sm Vs. Le with failures: A group of five applications is used for analysis. The completion times are shown in Table 5.18. Figure 5.18 shows a plot of the results.

Application	Ts_Sm	Le(Failures)
1	1293	1413
2	277	384
3	1086	1213
4	1081	1211
5	1038	1195

Table 5.18: Fault tolerant executions on a single machine:: Se Vs. Le with failures

We have to observe that for a failed application to get completed the overheads involved are fault detection and restoration. The time taken for fault detection depends on thee detection interval. If a particular machine sends a fault check message every x seconds and waits for a reply for y seconds, then the failure detection process takes utmost x+y seconds. A failure detection is followed by broadcast messages to update the virtual ting and then restoration process. In our environment, this process takes approximately 5 seconds.

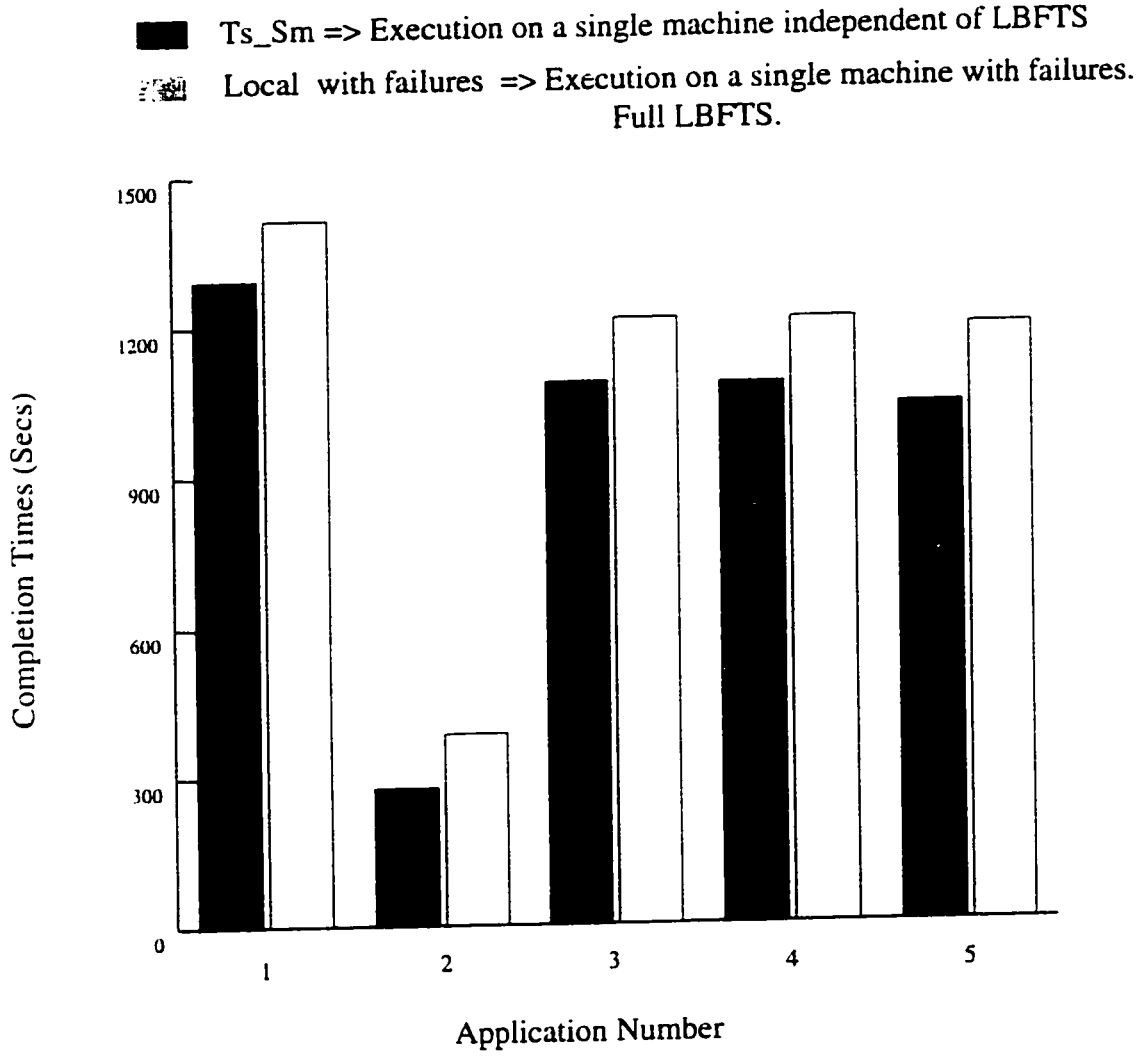


Figure 5.18: Restoration :: Ts_Sm Vs. Le with failures: Checkpoint 60 secs

The figure shows that all the processes are subjected to the same amount of overhead, about 2 minutes, when compared to the sequential execution. This extra time can be accounted for LBFTS daemon, checkpointing, failure detection and restoration. The reason behind this amount of time is because fault detection time(FDT) was chosen as 40 seconds and the reply interval(RI) was 30 seconds. If there is a fault the system takes atleast RI and atmost (FDT + RI) to identify the fault.

Ipcpm Versus Ipcpm with failures: The applications are run in a fully load balanced manner. In the first case there are no failures over the network. In the second case random failures are injected in the machines. Table 5.19 provides the results. Figure 5.19 shows a plot of the completion times for the failure and non-failure cases. From the results we can see that some applications have a large restoration time while some applications have less completion times in the failure case. As the system induces random failures each application has been subjected to more than one failure. More the number of failures for a particular application more the overhead taken for restoration process, hence the larger completion times. In other cases the applications have found a lightly loaded machine and hence the reduction in completion times. From this we can conclude that though the applications encounter failures some applications have better completion times.

Application	Se	Ip
1	436	690
2	105	76
3	521	696
4	296	203
5	119	184

Table 5.19: Fault tolerant execution:: Ipcpm Vs. Ipcpm with failures

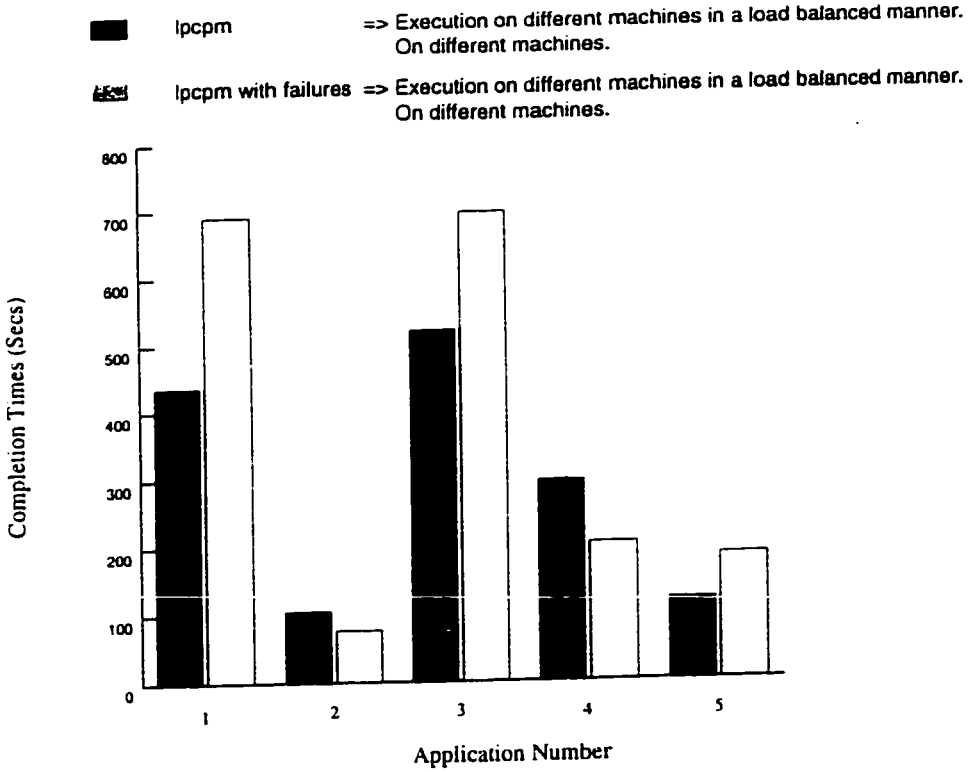


Figure 5.19: Restoration :: Ipcpm Vs. Ipcpm with failures: Checkpoint 60 secs

Chapter 6

Conclusions and Future work

In this research the primary objective has been to provide a load balanced and reliable environment in Unix based Network of Workstations. To realize this goal we have designed and developed a subsystem LBFTS (Load Balancing and Fault Tolerance Subsystem), which sits on other layers of OS. LBFTS paves the way for efficient utilization of system resources over the network while providing a fault tolerant environment for user applications in a user transparent manner. The achievements of this research can be summarized as below:

- Designed and developed a Load Balancing Subsystem which allows high utilization of workstations and improves completion times for certain long running applications.

- Designed and developed a Fault Tolerance Subsystem which allows applications to continue execution from the point they have failed.
- Incorporated Process Migration Subsystem into LBFTS which provides the migration capability for all types of applications.
- The system can handle non-communicating as well as communicating applications.
- Overhead due to LBFTS is negligible in the case of large applications.
- Average speed up of 2 is attained for a network of three workstations of same configurations.
- Initial placement without migration is sufficient for short lived applications.
- Migration can be applied to long running applications for better response times.
- Fault tolerance subsystem achieves reliability while incurring reasonable overhead.

Further exploration can be done in the following topics:

1. PMS is currently implemented within the kernel of LINUX. For portability reasons other alternatives have to be considered. PMS allows migration only

across equivalent/compatible machine architectures and versions of the OS. Providing migration across heterogeneous architectures is a major problem that can be pursued further.

2. LBFTS has been experimented for a small network. It has to be expanded for a large network and tested for speedup.

3. Performance of LBFTS under background load has not been studied. Such experiments have to be carried out on larger network of workstations in a real environment.

Bibliography

[1] M. Bozyigit, K Al-Tawil, and S.K. Naseer, "A task migration facility for parallel & distributed applications", *PDPTA International Conference*, 1997.

[2] S.K. Naseer, "A process migration subsystem for distributed applications", Master's thesis, KFUPM, 1995.

[3] W. Zhu, P. Socko, and B. Kiepuszewski, "Migration impact on load balancing– an experience on amoeba", *Operating Systems*, vol. 31, no. 1, pp. 43–53, January 1997.

[4] D. Arredondo, M. Erricalde, and S. Flores, "Load distribution and balancing support in a workstation based distributed system", *IEEE*, vol. 31, no. 1, pp. 54-63, January 1997.

[5] J.L. Michael, M. Livny, and M.W. Mutka, "Condor-a hunter of idle workstations", in *Eighth International Conference on Distributed Computing*, 1988, pp. 104-111.

[6] P. Krueger and R. Chawla, "The stealth distributed scheduler", *IEEE*, vol. 31, no. 1, pp. 54 63, January 1991.

[7] S. Zhou, "A trace driven simulation study of dynamic load balancing", *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1327-1341, September 1988.

[8] G.S. Wolf et.al, "An experimental study of workload indices for non-dedicated, heterogeneous systems", in *PDPTA International Conference*, 1997, pp. 470-478.

[9] M.H. Mair and A.P. Reeves, "Strategies for dynamic load balancing on highly parallel computers", *IEEE Transactions on Parallel and Distributed systems*, vol. 4, no. 9, pp. 979 993, September 1993.

[10] S.N. Haq, "A load balancing framework for distributed and parallel applications", Master's thesis, KFUPM, 1995.

[11] S.N. Haq, M. Bozyigit, S. Ghanta, and S.K. Naseer, "Design of a load balancing framework for distributed and parallel applications", *PDPTA International Conference*, vol. 3, 1997.

[12] M Bozyigit and M Melhi, "Load balancing framework for distributed systems", *Computer Systems Science & Engineering*, vol. 12, no. 5, pp. 287-293, September 1997.

[13] I.R. Philip, "Dynamic load balancing in distributed systems", in *IEEE proceedings 1990 southeastcon*, 1990, pp. 304 327.

[14] K. Erciyes and S. Yilmaz, "Dynamic load balancing in a distributed computer system", *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1579 -1586, December 1989.

[15] C. Hou and K.G. Shin, "Implementation od decentralized load sharing in networked workstations using the condor package", *Journal of parallel and distributed computing*, vol. 40, 1997.

[16] M.V. Deverakonda and R.K. Iyer, "Predictability of process resource usage : A measurement-based study on unix", *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1579–1586, December 1989.

[17] K.K. Goswami, R.K. Iyer, and M.V. Deverakonda, "Load sharing based on task resource prediction", in *22nd annual Hawaii Int. Conference system sciences*, January 1989, pp. 921 -927.

[18] K.K. Goswami, M.V. Deverakonda, and R.K.Iyer, "Prediction-based dynamic load-sharing heuristics", *IEEE Transactions on Parallel and Distributed systems*, vol. 4, no. 6, pp. 638–648, June 1993.

[19] T. Monteil and J.M. Garcia, "Task allocation strategies on workstations using processor load prediction", *PDPTA International Conference*, 1997.

[20] F. Tandiari, S.C. Kothari, A. Dixit, and E.W. Anderson, "Batrun:utilizing idle workstations for large scale computing", *IEEE Parallel and Distributed Technology*, 1996.

[21] M. Nuttal and M. Sloman, "Workload characteristics for process migration and load balancing", in *17'th International conference on DCS*, 1997, pp. 133–140.

[22] E.S. Leguizamon and M. Gallard, "A hybrid strategy for load balancing in distributed systems environments", in *International conference on Evolutionary Computation*, 1997, pp. 127–132.

[23] J. Zaki, W. Li, and S. Parthasarathy, "Customized dynamic load balancing for a network of workstations", in *Fifth International symposium on High Performance Distributed Computing*, 1996, pp. 282–291.

[24] A. Borg, W. Blau, W. Graetcsh, Wolfgang Oberle, and F. Herrmann, "Fault tolerance under unix", *ACM Transactions on Computer Systems*, vol. 7, no. 1, pp. 1–24, Febrauary 1989.

[25] Y. Artsy and R. Finkel, "Designing a process migration facility", *IEEE Computer*, 1989.

[26] F. Douglis and J. Ousterhout, "Transparent process migration: Design alternatives and the sprite implementation", *Software-Practice and Experience*, vol. 21, no. 8, pp. 757–785, August 1991.

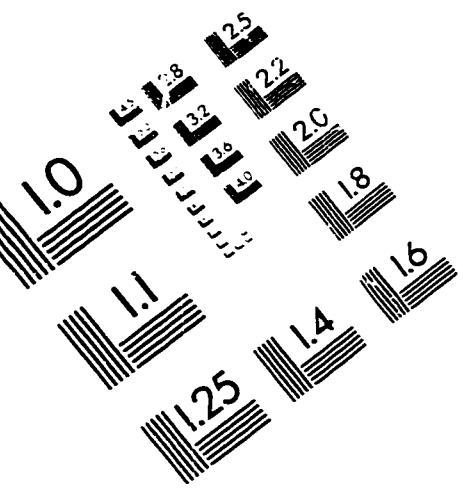
[27] M. Richmond and M. Hitchens, "A new process migration algorithm", *Operating Systems*, vol. 31, no. 1, pp. 31–42, January 1997.

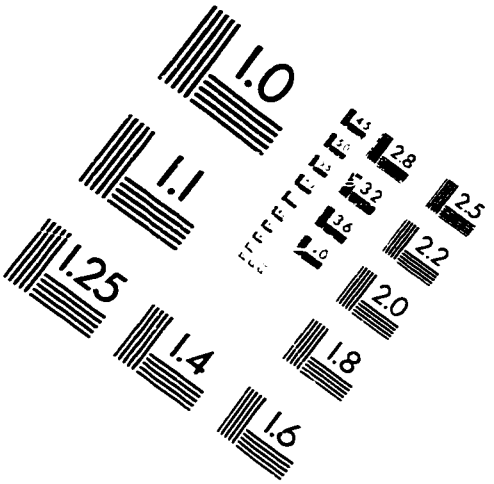
[28] J.S. Plank and K. Li, "ickp: A consistent checkpointer for multicomputers", *IEEE Parallel and Distributed Technology*, 1994.

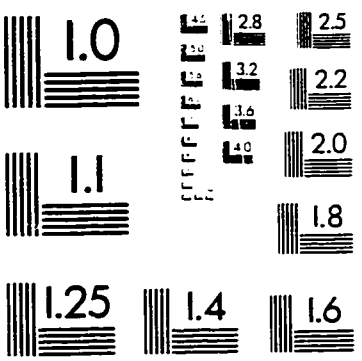
[29] R. Prakash and M. Singhal, "Low-cost checkpointing and failure recovery in mobile computing systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 10, pp. 1035–1043, October 1996.

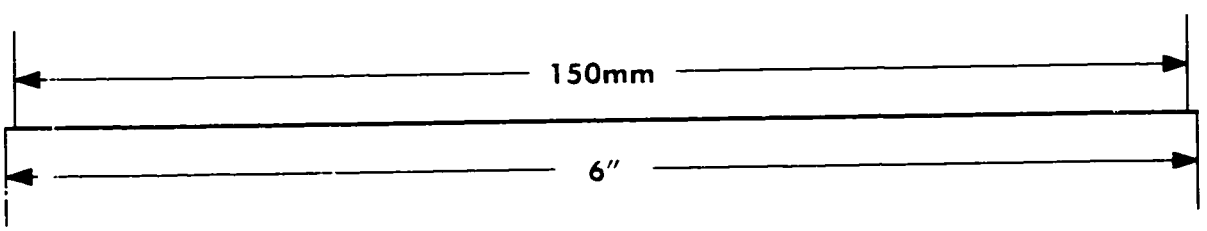
[30] A.B. Ramanovsky, "Coversational group service", *Operating Systems*, vol. 31, no. 1, pp. 54–63, January 1997.

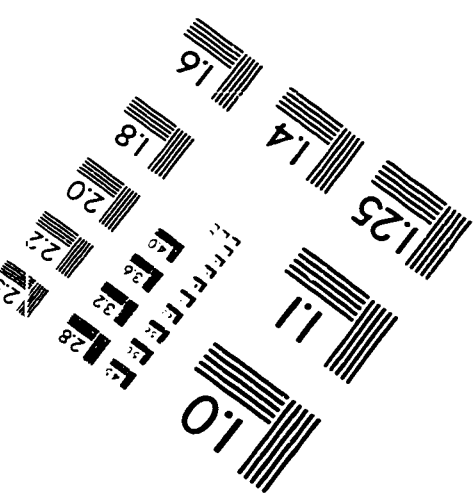
IMAGE EVALUATION
TEST TARGET (QA-3)

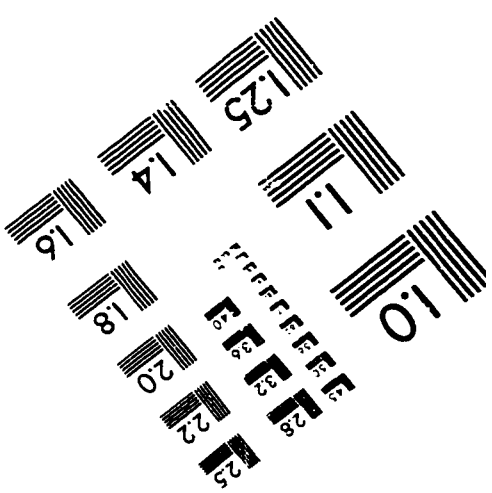












APPLIED IMAGE . Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989